The Sixteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications

Energy-Efficient Static Priority and Speed Assignment for Real-Time Tasks with Non-Deterministic Release Times

Simon Perathoner, Lothar Thiele
Computer Engineering and Networks Laboratory (TIK)
Swiss Federal Institute of Technology (ETH)
Zurich, Switzerland
Email: {perathoner,thiele}@tik.ee.ethz.ch

Jian-Jia Chen
Institute for Process Control and Robotics (IPR)
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
Email: jian-jia.chen@kit.edu

***Abstract*—Dynamic Voltage Scaling (DVS) has been widely used for decreasing the dynamic power dissipation of processors. For real-time systems, DVS techniques have been developed that permit to meet the timing constraints of multiple real-time tasks and at the same time reduce the overall dynamic energy consumption. Known methods for static priority DVS scheduling are, however, either restricted to simple periodic/sporadic task release patterns or presume full a priori knowledge of task release times. Moreover, none of the present approaches considers the optimization of task priorities for reducing the energy consumption. In this paper we explore how to determine the static priorities and individual execution speeds (supply voltages) of multiple tasks with non-deterministic release times bounded by arrival curves such that the energy consumption is reduced and the real-time constraints are met. The result are different heuristics for the design of DVS-based real-time systems with static priorities. We show that the proposed methodology leads to energy-efficient system designs and demonstrate the applicability of the approach by means of experiments.**

I. Introduction

Power dissipation increasingly constrains the performance of modern computing systems. Therefore, power management in both hardware and software has become one of the principal aspects of system design. Effective power management can significantly prolong the battery lifetime of embedded systems or reduce the power bill for server systems. Dynamic voltage scaling (DVS) and dynamic power management (DPM) have been introduced to trade system performance for energy consumption. For DVS processors, the dynamic energy consumption due to charging and discharging of CMOS gates can be reduced by means of a lower supply voltage and, consequently, a lower execution speed. Hence, DVS scheduling algorithms, e.g., [2], [25], tend to execute jobs as slowly as possible in order to save energy. On the other hand, to reduce the static energy consumption due to leakage, DPM scheduling algorithms, e.g., [10], tend to aggregate jobs and put the system to sleep mode when there are no jobs to execute.

For real-time systems, energy-efficient design stands for minimizing the energy consumption without violating the timing constraints of real-time tasks. Known techniques can broadly be divided into the two main classes of offline and online methods. The former determine the system behavior statically by assuming worst-case workload for tasks, see, e.g., [25]. The latter reclaim energy dynamically by adapting to the actual workload, see, e.g., [2].

Most studies for energy-efficient scheduling in real-time systems assume, however, that tasks are released periodically/sporadically or that the precise release pattern of tasks is known a priori. Unfortunately, for many practical systems with event-triggered tasks these assumptions are not realistic because precise arrival times of events cannot be predicted. There are two main reasons for non-determinism in the timing of the triggering event streams: (a) Tasks of embedded systems are often triggered by the physical environment which can, in general, not be predicted accurately (b) In distributed architectures tasks on different processing components trigger each other; variable execution workloads, communication delays, as well as interferences on shared resources make the prediction of precise event arrival times extremely complex. In the domain of performance analysis of distributed embedded systems powerful abstractions have been developed to capture timing non-determinism of event streams. Examples are event models such as periodic with jitter/burst or arrival curves which are employed by the performance analysis methods SymTA/S [9] and Real-Time Calculus (RTC) [23]. The DVS algorithms that we propose in this paper are based on RTC and use arrival curves, an abstraction for the characterization of *arbitrary* event streams. This leads to a considerably larger modeling scope compared to most present DVS techniques.

In this paper we consider the energy-efficient static priority preemptive scheduling of multiple real-time tasks with non-deterministic release times. We assume that the release times are not known a priori but constrained by arrival curves. For this setting, it is still an open issue how to statically or dynamically determine the execution speeds (supply voltages) and the priorities of the individual tasks such that the energy consumption is minimized. We focus on offline algorithms, that is, on methods that fix the execution parameters at design time according to the predicted worst-case workload. In particular, we explore how to statically determine the priorities and execution speeds of multiple tasks that are triggered by arbitrary event streams. We first consider systems with a bounded continuous speed range, and then adapt the developed concepts for systems with discrete operating speeds.

Determining the optimal assignment of individual execution speeds in a static priority setting and under arbitrary release patterns is not trivial. For instance, it is often not wise to execute a high priority task as slowly as possible as this

1533-2306/10 $26.00 © 2010 IEEE
DOI 10.1109/RTCSA.2010.9

might force low priority tasks to use very high speeds in order to meet their timing constraints. Also, there is a mutual dependency between priorities and execution speeds: a task can tolerate a lower execution speed if it gets a higher priority. To the best of our knowledge, all present DVS methods for static priority systems consider task priorities as given and hence preclude an important possibility to further reduce the energy consumption.

This paper provides the following contributions:

- We devise a simple algorithm for computing a static priority assignment to multiple real-time tasks with arbitrary release patterns. We show that the computed priority assignments are energy-optimal for the particular case that one global speed is used to execute all the tasks.
- We show that priority-monotonic speed assignments are energy-efficient and present a heuristic that determines the individual task execution speeds in the particular case that the task priorities are predetermined.
- We propose an efficient heuristic for the general case in which both priorities and execution speeds need to be determined.
- We demonstrate the effectiveness of the presented methods on several experimental test cases.

II. Related work

The need for energy-efficient real-time computing systems has driven research for many years. As a result, there is a large body of results on DVS and DPM scheduling techniques for real-time systems. The existing approaches can coarsely be classified into offline and online methods. Offline algorithms, e.g., [19], [25], [26], take scheduling decisions statically (at design time) according to the expected worst-case workload of tasks. Online algorithms, e.g., [2], [18], decide the task scheduling dynamically (at run time) by adapting to the actual workload. While online approaches can help to considerably reduce the energy consumption in the case of light workloads, they often perform worse than offline methods for heavy workloads, see, e.g., [25]. In addition, in order not to harm any real-time constraints, the feasibility of all dynamic decisions has to be carefully verified a priori [5].

Most of the work on DVS scheduling for real-time systems assumes that tasks are scheduled according to the Earliest Deadline First (EDF) priority assignment, e.g., [2], [12], [25]. There are, however, also several results for static priority scheduling which we will briefly discuss in the following. The authors of [26] propose an offline method for the computation of a near-optimal voltage schedule assuming full a priori knowledge of task release times. In [19] a more efficient solution to the same problem is provided. Other approaches for static priority DVS scheduling assume periodic/sporadic task releases, e.g, [18], [21]. In [21] the effect of a limited number of operating frequencies on the performance of voltage-scaling algorithms is studied and four different voltage scheduling schemes are presented. One of the four heuristics relies on priority-monotonic speed assignments, which is also the central idea of our work. The authors of [18] take into account transition overheads for speed changes, while in [8] energy-efficient preemption thresholds are discussed.

Note that almost all present methods for static priority DVS scheduling are based on deterministic task release patterns, meaning that they either assume a priori knowledge of task release times or presume a periodic (or at most sporadic) release pattern. Two exceptions are the methods proposed in [22] and [20]. The authors of [22] consider mixed task sets (periodic and aperiodic tasks) and use scheduling servers to handle aperiodic task activations. They discuss interesting tradeoffs between energy consumption and response times of aperiodic tasks, however, cannot guarantee the schedulability of aperiodic tasks. In [20] the authors admit limited non-determinism for task release times and express it by means of a periodic with jitter event model. They provide a stochastic method for power optimization based on multi-dimensional evolutionary algorithms. All mentioned approaches for static priority DVS scheduling assume given task priorities and do not consider priority assignment for energy reduction.

Power-aware processing of tasks with non-deterministic release times bounded by arrival curves has been recently explored in [5], [17]. For systems processing a single task, the authors of [17] explore how to choose the minimal execution speed that avoids an overflow of the input buffer, while in [5] a framework is proposed to analyze the feasibility of online DVS algorithms. To the best of our knowledge, our work is the first contribution to DVS scheduling of multiple tasks with non-deterministic release times.

The problem of assigning static priorities to tasks executed by a preemptive scheduler has also been studied. For periodic/sporadic tasks optimality results are available with respect to schedulability, e.g., for rate-monotonic (RM) priorities [16] and deadline-monotonic (DM) priorities [15]. RM and DM cease to be optimal in the case of asynchronous task release, i.e., if the tasks have arbitrary offsets. The authors of [1] provide an optimal priority assignment method for this more general case. Also several other extensions have been worked out for issues such as execution blocking [4], task re-executions [7], and limited priority levels [3]. Note that all present approaches for priority assignment assume strictly periodic tasks and hence exclude any non-determinism in the release patterns except for the offsets. This is not the case for [6] where the authors determine the optimal priorities for arbitrary released aperiodic tasks in a mixed task set, but again without schedulability guarantees. Note further, that all mentioned approaches optimize priorities with respect to schedulability. We are not aware of any methods for real-time systems that optimize priorities with respect to energy consumption, as proposed in the present work.

III. Models and Problem Definition

In this section we first describe the system and power models adopted in this paper. Then, we define the considered problem and present a motivational example.

A. System Model

The real-time system that we are studying consists of a processor that executes N independent tasks, $\Gamma = \{\tau_1, \cdots, \tau_N\}$, according to a preemptive scheduling policy with static priorities. The processor implements DVS scheduling and can set different execution speeds (supply voltages) at different times. For the sake of simplicity, we assume that there is no overhead for speed changes. We denote the set of available execution speeds with $\hat{\mathcal{S}}$. Modern DVS processors have only a discrete set of available speeds. However, one can apply voltage hopping at the software level [14] in order to make continuous speeds available. In this paper we consider both cases, i.e., we either assume $\hat{\mathcal{S}} = [\hat{s}_{min}, \hat{s}_{max}]$, or $\hat{\mathcal{S}} = \{\hat{s}_1, \hat{s}_2, \cdots, \hat{s}_K\}$ with $\hat{s}_{min} := \hat{s}_1, \ \hat{s}_{max} := \hat{s}_K, \ \hat{s}_1 < \hat{s}_2 < \cdots < \hat{s}_K$, and discuss the two cases separately in Sec. V. Without loss of generality, we consider $\hat{s}_{max} = 1$ and normalize all related metrics.

Each task τ_i is characterized by the following parameters:

- $\bar{\alpha}_i(\Delta)$: Arrival curve bounding the (non-deterministic) release pattern
- C_i : Execution time at the maximum speed $\hat{s}_{max}$
- D_i : Deadline (relative to the task release)

The arrival curve $\bar{\alpha}_i(\Delta)$ bounds the maximum number of task releases in *any* time interval of length Δ. Since $\bar{\alpha}_i$ is a function on the time-interval domain, it does not specify actual release times for τ_i, but only an upper bound on its non-deterministic release behavior. For more details on arrival curves, we refer the reader to Sec. IV. The worst-case response time WCRTτ_i of a task τ_i is defined as the longest time between the release and the completion of a task. A task τ_i is said to be schedulable if WCRT$\tau_i \leq D_i$.

B. Power Model

We consider the power model of [27], in which the power consumption of the processor at speed $\hat{s}$ is specified as follows:

$$P(\hat{s}) = P_{sta} + \hbar(P_{ind} + P_d) = P_{sta} + \hbar(P_{ind} + C_{ef}\hat{s}^{\gamma}) \quad (1)$$

In the above equation, P_{sta}, P_{ind}, and P_d are *static power*, *speed-independent* active power, and *speed-dependent* active power, respectively. If the system is in active mode, $\hbar$ is 1, whereas $\hbar$ is set to 0 when the system is in sleep mode. The constants C_{ef} and $2 \leq \gamma \leq 3$ are system-dependent and represent the effective switching capacitance and the dynamic power exponent, respectively.

We assume an architecture with an excessive time/energy overhead for turning the system off/on. Hence, we exclude the possibility to turn the system off dynamically. This means that the static power P_{sta} is not manageable, and therefore we do not need to consider it for the reminder of the paper. Instead, we focus on how to manage the active power consumption of the system. We assume that additionally to the DVS scheduling carried out in the active mode, the processor can switch to a sleep mode. In particular, we consider the simplest form of DPM in which the processor goes to sleep mode without active power consumption whenever it is idle, and returns to active mode as soon as a task is released. We assume that there is no overhead in terms of time or energy for the active/sleep mode switches. This is reasonable, if gated supply voltage is applied [27]. Under the mentioned assumptions, the (dynamic) energy consumption of the processor is merely a convex function of the execution speed and there is a *critical speed* $\hat{s}_{crit} \in \hat{\mathcal{S}}$ such that executing at $\hat{s}_{crit}$ is more energy-efficient than executing at any lower speed [11], [27]. By the definition of critical speed, only speeds in the set $\mathcal{S}$ should be used, where $\mathcal{S} = \{s \in \hat{\mathcal{S}} \mid s \geq \hat{s}_{crit}\}$. The set $\mathcal{S}$ can again be continuous, $\mathcal{S} = [s_{min}, s_{max}]$, or discrete, $\mathcal{S} = \{s_1, s_2, \cdots, s_M\}$ with $s_{min} := s_1, \ s_{max} := s_M, \ s_1 < s_2 < \cdots < s_M$.

C. Problem Definition

The problem tackled in this paper is to statically assign priorities and execution speeds to real-time tasks with bounded non-deterministic release patterns, such that all tasks are guaranteed to meet their deadlines and the worst-case energy consumption of the system is minimized. Formally, we can define the problem as follows:

Given is a set of real-time tasks $\Gamma = \{\tau_1, \cdots, \tau_N\}$ characterized as described above. Let $\Pi : \Gamma \rightarrow \{1, \cdots, N\}$ be a bijective function that assigns a unique priority to each task, where task τ_i has higher priority than task τ_j if $i < j$. Let $\Sigma : \Gamma \rightarrow \mathcal{S}$ be a function that assigns an execution speed to each task. Let $E(\Delta)$ denote the worst-case dynamic energy consumption of the system for any time interval of length Δ. The problem is to find Π and Σ such that the following two conditions hold:

- WCRT$\tau_i \leq D_i \quad \forall i \in \{1, \cdots, N\}$
- $E(\Delta)$ is minimized

In the defined setting, each task is executed with a *constant* speed, meaning that speed changes are allowed only at context switches. The constant execution speed of a task has to be chosen such that the worst-case workload of the task can be handled at any point in time. Note that a more fine-grained DVS schedule in which different instances of the same task are executed with different speeds, or in which the speed is changed *during* the execution of a task instance, see e.g. [24], makes sense only in the presence of a priori knowledge of task release times, or in the context of online DVS.

Note further that for different time intervals Δ_1, Δ_2 different static speed assignments Σ_1, Σ_2 can be optimal in terms of energy consumption, as we will show later. Hence, an overall optimal static speed assignment may not exist for a system. For this reason, in this paper we do not focus on the energy optimization for a given time interval Δ, but rather devise heuristics that determine reasonable solutions for all sufficiently large intervals.

D. Motivational example

We will now illustrate the above problem by means of a simple example. Consider a system consisting of three tasks τ_1, τ_2, τ_3 that are released periodically with some non-deterministic but bounded release jitter. The tasks are specified by the parameter tuples $\tau_1 = \langle 10, 3, 1, 1\rangle$, $\tau_2 = \langle 5, 3, 1, 9\rangle$, $\tau_3 = \langle 8, 1, 1, 10\rangle$. Each tuple contains the parameters $\langle p, j, C, D\rangle$

(expressed in ms) with the following meaning: p = period, j = max. jitter, C = execution time at s_{max}, D = deadline. Given these parameters, we want to determine the static priority and speed assignments for the tasks that minimize the long-term worst-case energy consumption of the system. The active power consumption of the CPU is assumed to be $P(s) = \hbar(0.08 + 1.52s^3)$ Watt.

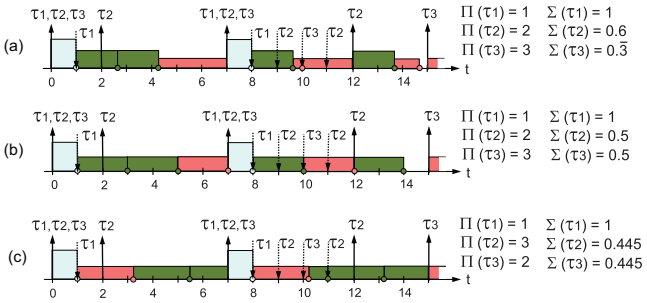

Fig. 1: Execution trace

Fig. 1 shows the system execution under worst-case workload (critical instant) for three different priority and speed assignments. In the figure, the upward and downward arrows indicate task releases and task deadlines, respectively. The rectangles represent the execution of task instances (also denoted as jobs). The height of a rectangle indicates the speed at which the corresponding job is executed. A small dot on the axis indicates the completion of a job. If no dot is shown at the bottom right of a rectangle, it means that the corresponding job is preempted by a higher-priority job. The respective priority and speed assignments are shown next to the execution traces.

A feasible priority and speed assignment is reported in Fig. 1(a). The corresponding execution trace shows that the first job of τ_3 completes just in time (at $t = 10$). We conclude that the speed assignment in (a) is 'tight', in the sense that lowering the execution speed for any of the three tasks would lead to a deadline violation. Nevertheless, we will show that the speeds chosen in (a) are not the most energy-efficient ones. In order to illustrate this, we compute the worst-case dynamic energy consumption of the system for a time interval of $10s$. By a simple simulation of the worst-case execution trace we obtain an energy consumption of $E_a(10s) = 3.475J$. Consider now the speed assignment of Fig. 1(b). It is again tight, as lowering any of the execution speeds would lead to a deadline violation for τ_3. For this case we compute a worst-case energy consumption of $E_b(10s) = 3.357J$, i.e., assignment (b) is more energy-efficient than (a). Finally, we decide to invert the priorities of τ_2 and τ_3. Fig. 1(c) shows that this permits to use even lower speeds compared to (b), while the real-time constraints are still guaranteed. The corresponding worst-case energy consumption amounts to $E_c(10s) = 3.163J$. In Sec. V we will discuss heuristics for finding such energy-efficient priority and speed assignments.

IV. Analysis Framework

The algorithms proposed in this paper are based on Real-Time Calculus (RTC) [23], a general framework for scheduling analysis that has its roots in Network Calculus [13]. In the following, we will briefly introduce the basic abstractions of RTC that are employed in this paper.

A. Workload Model

The release pattern of a task can be represented by means of a timed event stream where each event in the stream corresponds to the release of a new task instance. In RTC timed event streams (task release patterns) are abstracted by means of *arrival curves*. An arrival curve $\bar{\alpha}(\Delta) \in \mathbb{R}^{\geq 0}$, $\Delta \in \mathbb{R}^{\geq 0}$ specifies an upper bound on the number of events in *any* time interval of length Δ. Note that arbitrary task release patterns can be represented by means of arrival curves. They generalize classical representations of event streams such as sporadic, periodic or periodic with jitter. For instance, for a periodic stream with jitter that is characterized by period p, maximum jitter j and minimum event inter-arrival time d (also denoted as PJD-stream) the arrival curve is given by

$$\bar{\alpha}(\Delta) = \min\left\{\left\lceil\frac{\Delta + j}{p}\right\rceil, \left\lceil\frac{\Delta}{d}\right\rceil\right\}. \quad (2)$$

An example of arrival curves is shown in Fig. 2(a). The figure represents the arrival curves that bound the task release patterns in the example system of Sec. III-D.

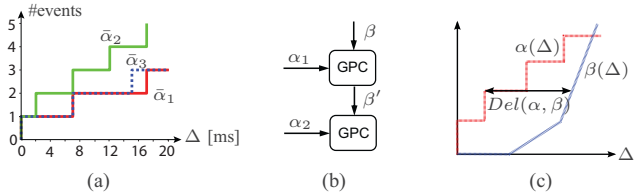

Fig. 2: (a) Arrival curves for the motivational example. (b) GPC and model of preemptive FP scheduling. (c) Graphical interpretation of Eq.(3).

Note that an event-based arrival curve $\bar{\alpha}$ can be converted to a workload-based arrival curve α by scaling it with the execution time of the corresponding task. We assume that all instances of a task have the same execution time C at the maximum speed s_{max}. The curve $\alpha(\Delta) = C \cdot \bar{\alpha}(\Delta)$ denotes an upper bound for the processing demand of the task in any time interval of size Δ. The processing demand is expressed in units of execution time at speed s_{max}. If the task is executed at a different speed $s \in \mathcal{S}$, the processing of an instance takes C/s time units. Hence, the curve $\alpha(\Delta)/s$ denotes an upper bound for the execution time demanded by the task in any time interval of size Δ, under the assumption that it is executed at speed s.

B. Resource Model

A similar abstraction, so-called *service curves*, are used to characterize the availability of processing resources to a task. In particular, we use a service curve $\beta(\Delta) \in \mathbb{R}^{\geq 0}$, $\Delta \in \mathbb{R}^{\geq 0}$ to specify a *lower* bound on the available execution time in *any* time interval of length Δ.

C. Processing Model and Scheduling Analysis

In the framework of RTC, tasks that are executed on a (shared) processor are modeled by abstract processing components that manipulate arrival and service curves. An example of such an abstract component is the *Greedy Processing Component* (GPC), shown in Fig. 2(b). It models a task that is triggered by the events of an incoming event stream which queue up in a FIFO buffer. The task processes the events in a greedy fashion, while being restricted by the availability of processing resources. The workload imposed on the task by the input event stream is abstracted by an arrival curve α. The availability of processing resources is captured by a service curve β. The maximum delay experienced by an input event at a GPC is upper bounded by

$$\text{Del}(\alpha, \beta) \stackrel{def}{=} \sup_{\lambda \geq 0} \{\inf\{\tau \geq 0 : \alpha(\lambda) \leq \beta(\lambda + \tau)\}\}. \quad (3)$$

This delay corresponds to the WCRT of the modeled task. Fig. 2(c) shows the graphical interpretation of $\text{Del}(\alpha, \beta)$. A task with workload bound α and relative deadline D is said to be *schedulable* if $\text{Del}(\alpha, \beta) \leq D$, that is, if the WCRT of the task is not larger than its deadline. We call a task set Γ schedulable if every task $\tau \in \Gamma$ is schedulable.

The amount of processing resources left over by a GPC is bounded by the service curve

$$\beta'(\Delta) = \sup_{0 \leq \lambda \leq \Delta} \{\beta(\lambda) - \alpha(\lambda)\} \stackrel{def}{=} \text{RT}(\beta, \alpha). \quad (4)$$

Note that the above bound is tight, meaning that RT is a lossless transformation. Scheduling policies for the execution of multiple tasks on a shared processor are modeled by appropriately connecting abstract processing components. For instance, Fig. 2(b) shows the RTC model for the preemptive static priority scheduling of two tasks, where the lower priority task only gets the processing resources left over by the higher priority task.

V. Proposed Algorithms

This section presents our proposed algorithms for the static priority and speed assignment problem. For the sake of clarity, we will first discuss the assignments of priorities and speeds in isolation (Sec. V-A and V-B), and then show how the algorithms can be integrated to achieve more energy-efficient solutions (Sec. V-C). We will start with considering systems with continuous speeds, and then explain how the algorithms need to be adapted for the discrete case (Sec. V-D).

A. Priority Assignment

Let us first consider the simple case where we want to determine the priority assignment Π for a task set Γ with *fixed* speed assignment Σ. Since the execution speeds for the tasks are fixed, so is also the worst-case energy consumption of the system. This is because the amount of energy consumed depends only on the speed at which the single tasks are executed, but not on the order in which the tasks are processed. Nevertheless, the priorities of the single tasks have an influence on the schedulability of the system. In particular, we want to determine a priority assignment Π under which Γ is schedulable, if such an assignment exists. Note that if N is the number of tasks in Γ, then there are $N!$ possible priority assignments, and a brute-force algorithm would have to check all of them in the worst case. For particular classes of systems, efficient solutions to this problem can be found in the literature. For instance, the deadline-monotonic (DM) priority assignment has been shown to be optimal for periodic tasks with deadlines that do not exceed the respective periods [15]. In the following, we devise a simple and efficient algorithm that solves the priority assignment problem in the general case, i.e., under arbitrary task activation patterns.

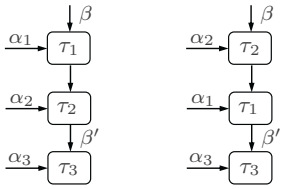

Fig. 3: Two chains of GPCs with different priority assignments but same service for τ_3

The basic idea behind the method is that the amount of processing resources that is available to a task does not depend on the order in which higher-priority tasks are processed. This is illustrated in the example of Fig. 3, where the input service curve β' of the lowest-priority task τ_3 is the same in both shown cases. Hence, in order to verify whether a given task is schedulable at the lowest free priority level, it is sufficient to perform a single run of the scheduling analysis with an arbitrary arrangement of the higher-priority tasks. This concept is employed iteratively in Alg. 1 in order to derive a schedulable priority assignment Π for a task set Γ that is executed on a processor with service β. The algorithm traverses all priority levels starting from the lowest

Algorithm 1 `Determine priorities`

```
1: function FIND_SCHEDULABLE_ORDER(Γ, Σ, β): Π
2:     Π ← random priority assignment on Γ
3:     for i ← N, 1 do
4:         last_task_schedulable ← false
5:         for j ← 1, i do
6:             Π′ ← Π
7:             Π′(Π^{-1}(i)) ← j
8:             Π′(Π^{-1}(j)) ← i
9:             β′ = β
10:            for k ← 1, i − 1 do
11:                τ_x ← Π′^{-1}(k)
12:                β′ = RT(β′, α_x/Σ(τ_x))
13:            end for
14:            τ_x ← Π′^{-1}(i)
15:            if Del(α_x/Σ(τ_x), β′) ≤ D_x then
16:                last_task_schedulable ← true
17:                Π ← Π′
18:                break
19:            end if
20:        end for
21:        if not last_task_schedulable then
22:            Warning(Γ NOT SCHEDULABLE)
23:            return ⊥
24:        end if
25:    end for
26:    return Π
27: end function
```

and assigns a task to each level. The task is chosen among the unassigned ones by making sure that it will be schedulable at the given priority level. In the algorithm we make use of the inverse priority assignment function $\Pi^{-1}(\gamma)$, which for a given priority level γ returns the task assigned to it. Note that the time complexity of Alg. 1 is $O(N^3)$. The following theorem formulates the optimality of the described priority assignment strategy.

Theorem 1: Let Γ be a task set characterized as described above and Σ a speed assignment for Γ. Let β be the service guaranteed by a processor with preemptive static priority scheduling. Then, Alg. 1 finds a priority assignment Π such that the system $\langle \Gamma, \Pi, \Sigma, \beta \rangle$ is schedulable, provided that such a priority assignment exists.

Proof: (Sketch) The above theorem can be shown by contradiction. Assume that $\exists \Pi$ such that $\langle \Gamma, \Pi, \Sigma, \beta \rangle$ is schedulable, but Alg. 1 does not return Π. This means that Alg. 1 either returns another schedulable priority assignment Π', which is fine, or it returns $\perp$. In the latter case, the algorithm must have reached a priority level γ such that none of the γ unassigned tasks can tolerate the execution at a lower priority level than all the remaining $\gamma - 1$ tasks. Note that revising previously done priority assignments is not helpful, as this would comport an even lower priority for at least one of these γ tasks. Hence, Alg. 1 returns $\perp$ only in the case that there is no schedulable priority assignment for the system. ■

The above algorithm can be extended to solve another simplified instance of the problem described in Sec. III-C, namely the case that we want to determine the most energy-efficient pair (Π, Σ) for a task set Γ, under the restriction that all tasks execute at *equal* speed. In other words, we can determine the priority assignment that allows the minimum *global* execution speed. The proposed solution is shown in Alg. 2 and will be reused later within the combined optimization heuristic presented in Sec. V-C. In Alg. 2 we use the notation Σ_s to refer to the assignment of the same speed s to all tasks in Γ. The algorithm follows a simple binary search strategy, in which the global speed is increased if no schedulable priority assignment is found or decreased otherwise. The search is stopped once a specified precision ϵ is reached for the approximation of the minimum global speed. Alg. 2 has time complexity $O(N^3 \log \frac{1}{\epsilon})$.

Corollary 1: Let Γ be a task set characterized as described above. Let β be the service guaranteed by a processor with preemptive static priority scheduling and let $\mathcal{S} = [s_{min}, s_{max}]$ denote the continuous set of available execution speeds. Assume that a priority assignment Π and a *minimal* speed $s \in \mathcal{S}$ exist such that $\langle \Gamma, \Pi, \Sigma_s, \beta \rangle$ is schedulable. Let $\epsilon > 0$ be an arbitrary small constant. Then, Alg. 2 finds a priority assignment Π' and a speed $s' \in \mathcal{S}$ with $s' - s \leq \epsilon$ such that the system $\langle \Gamma, \Pi', \Sigma_{s'}, \beta \rangle$ is schedulable.

Proof: (Sketch) Follows directly from Theorem 1 and the monotonicity of the schedulability with respect to the execution speed: A task set Γ schedulable at a global speed s is schedulable also at a global speed $\tilde{s} > s$. ■

Algorithm 2 `Determine min global speed`

```
1: function MIN_GLOBAL_SPEED(Γ, β, S, ε): s, Π
2:     Π ← Find_schedulable_order (Γ, Σ_smax, β)
3:     if Π = ⊥ then
4:         Error(Γ NOT SCHEDULABLE)
5:     end if
6:     s_up ← s_max
7:     s_lo ← s_min
8:     repeat
9:         s ← (s_lo + s_up)/2;
10:        Π' ← Find_schedulable_order (Γ, Σ_s, β)
11:        if Π' = ⊥ then
12:            s_lo ← s
13:        else
14:            s_up ← s
15:            Π ← Π'
16:        end if
17:    until s_up − s_lo ≤ ε
18:    return s_up, Π
19: end function
```

Since the power consumption function is monotone on $\mathcal{S}$, we conclude that Alg. 2 finds the energy-optimal *global* speed assignment that guarantees the timing constraints.

B. Speed Assignment

Let us now consider the opposite case in which we want to determine an energy-efficient speed assignment Σ for a task set Γ with *fixed* priority assignment Π. For the sake of simplicity, in the present subsection we will make use of the abbreviated notation $\tau^i_{pr} := \Pi^{-1}(i)$ to refer to the task assigned to priority level i. The task parameters are denoted accordingly: $\alpha^i_{pr}, D^i_{pr}, \cdots$. Similarly, we denote the execution speed of the task at priority level i with $s^i_{pr} := \Sigma(\Pi^{-1}(i))$, and the service available to that task with β^i_{pr}.

In order to minimize the energy consumption for the execution of N tasks with static priorities, it is not always adequate to assign the same execution speed to all the tasks, as done in Sec. V-A. For instance, the minimum acceptable global speed for a task chain could be determined by a high priority task with a stringent deadline. However, this speed could be unnecessarily high for tasks with lower priorities and less stringent timing requirements. Considering the convexity of the power function, one should further slow down lower priority tasks as much as possible and avoid premature task completions. In other words, one should always use *tight* speed assignments, where we call an assignment tight if slowing down any of the speeds in the assignment comports a deadline violation in the system. Nevertheless, how to choose best the individual execution speeds in a tight speed assignment is not trivial. This is because the assignment of a speed to a particular priority level affects the speeds required at lower priority levels. For instance, it might not be energy-efficient to execute a high priority task very slowly, if in turn this forces a low priority task to use a very high speed to meet its deadline. The convexity of the power function suggests to use the same execution speed for both tasks in such a situation, or as close speeds as possible if two different speeds are necessary. This is illustrated in the example of Fig. 4, which shows the execution traces of a system with two tasks under different speed assignments. Trace (a) results from an

assignment with $s^1_{pr} < s^2_{pr}$. Assume that the assignment is tight with respect to the deadline of τ^2_{pr}. Trace (b) results from another tight assignment where $s^1_{pr} = s^2_{pr}$. This second assignment is more-energy efficient due to the convexity of the power function. Note that in (b) we have increased s^1_{pr} (which certainly does not harm the timing constraint of τ^1_{pr}) such that a constant speed can be used for both tasks.

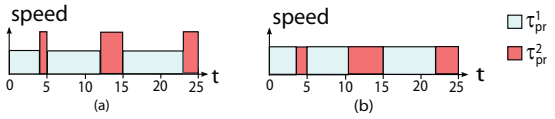

Fig. 4: Schedules in a busy interval. (a) Original schedule. (b) Lower energy consumption.

Based on the above observations, we propose to use speed assignments Σ that satisfy the following two properties:

$$s^1_{pr} \geq s^2_{pr} \geq \cdots \geq s^N_{pr} \tag{5}$$

$$s^{i-1}_{pr} > s^i_{pr} \text{ only if WCRT } \tau^{i-1}_{pr} = D^{i-1}_{pr} \tag{6}$$

The first property imposes a *priority-monotonic* speed assignment, in which a lower priority task never executes at a higher speed than a higher priority task. The second property enforces the use of as close speeds as possible by stating that a high priority task is assigned a higher speed than a low priority task only in the case that it cannot be further slowed down. Note that priority-monotonic speed assignments have also been explored in [21], however for strict periodic real-time tasks, only.

In the following we present a heuristic for the computation of energy-efficient speed assignments Σ that fulfills the above properties. The concept of the heuristic is to first determine a minimum *global* speed assignment for all tasks and then to find the 'bottleneck', that is, the task τ^i_{pr} with highest priority that cannot accept a lower speed assignment for any task τ^j_{pr} with $j \leq i$ because this would compromise its schedulability. At this point we make the speed assignments for $\tau^1_{pr}, \cdots, \tau^i_{pr}$ definitive and repeat the procedure for the tasks with lower priorities. The procedure stops once τ^N_{pr} becomes the bottleneck of a speed assignment.

Alg. 3 implements the described approach. The function `MinSpeed` adopts a binary search strategy in order to find the minimum feasible global speed for the remaining part of the task chain, given the service β_{in} left over by higher priority tasks. The schedulability test for the corresponding tasks is carried out by the function `Sched`, which implements Eq. 3. The function `FindBottleneck` determines the bottleneck of a given sub-chain of tasks with guaranteed service β_{in} and under constant speed s. Note that Alg. 3 has time complexity $O(N^2 \log \frac{1}{\epsilon})$.

Remarks on Non-Optimality: As pointed out in the problem description, for different time intervals different static speed assignments can be optimal in terms of energy consumption. Hence we cannot expect the above static algorithm to determine speed assignments that are optimal for arbitrary time intervals. In the following, we provide a brief example that highlights the potential non-optimality of the proposed speed assignment strategy.

Algorithm 3 `Determine speeds`

```
1: function DETERMINE_SPEEDS(Γ, Π, β, S, ε): Σ
2:     if SCHED(Γ, Π, 1, n, s_max, β) = true then
3:         s^0_pr ← s_max
4:         β^1_pr ← β
5:         i ← 0
6:         repeat
7:             i ← i + 1
8:             s ← MINSPEED (Γ, Π, i, N, s_min, s^{i-1}_pr, β^i_pr, ε)
9:             for k ← i, N do
10:                s^k_pr ← s
11:                β^{k+1}_pr = RT(β^k_pr, α^k_pr/s)
12:            end for
13:            if s − s_min < ε then
14:                break
15:            else
16:                i ← FINDBOTTLENECK (Γ, Π, i, s, β^i_pr, ε)
17:            end if
18:        until i = N
19:    else
20:        ERROR(UNSCHEDULABLE)
21:    end if
22:    return Σ = {τ^1_pr → s^1_pr, ··· , τ^N_pr → s^N_pr}
23: end function

24: function MINSPEED(Γ, Π, From_pr, To_pr, s_lo, s_up, β_in, ε): s
25:    repeat
26:        s ← (s_lo + s_up)/2;
27:        if SCHED(Γ, Π, From_pr, To_pr, s, β_in) = true then
28:            s_up ← s;
29:        else
30:            s_lo ← s;
31:        end if
32:    until s_up − s_lo ≤ ε
33:    return s_up
34: end function

35: function SCHED(Γ, Π, From_pr, To_pr, s, β_in): schedulable
36:    β' = β_in
37:    for k ← From_pr, To_pr do
38:        if Del(α^k_pr/s, β') ≤ D^k_pr then
39:            β' = RT(β', α^k_pr/s)
40:        else
41:            return false
42:        end if
43:    end for
44:    return true
45: end function

46: function FINDBOTTLENECK(Γ, Π, From_pr, s, β_in, ε): pr
47:    s_red ← s − ε
48:    for k ← From_pr, N do
49:        if SCHED (Γ, Π, From_pr, k, s_red, β_in) = false then
50:            return k
51:        end if
52:    end for
53: end function
```

Example 1: Consider the execution of two periodic tasks specified by the parameter tuples $\tau^1_{pr} = \langle 8, 0, 1, 2\rangle$ and $\tau^2_{pr} = \langle 4, 0, 1, 4\rangle$ (The format of the tuples is again $\langle p, j, C, D\rangle$). Assume that τ^1_{pr} is executed at a general speed $s^1_{pr} = x \geq 0.5$, as shown in the worst-case arrival trace represented in Fig. 5. Consequently, τ^1_{pr} needs to be executed at speed $s^2_{pr} = \frac{1}{4-1/x}$ in order to meet its deadline. Assume that the speed-dependent power consumption of the system is simply given by $P(s) = s^3$. Then, the energy consumption of the system for the interval Δ_2 indicated in Fig. 5 is given by $E_x(\Delta_2) = \frac{1}{x} \cdot x^3 + 2 \cdot (4 - \frac{1}{x}) \cdot (\frac{1}{4-1/x})^3$, which is minimized for $x = 0.565$. Alg. 3 would, however, determine $x = 0.5$,

that is, execute both tasks at the same speed $s^1_{pr} = s^2_{pr} = 0.5$, which minimizes the energy consumption in Δ_1.

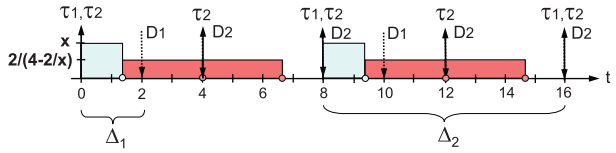

Fig. 5: Execution trace for example system

The above example demonstrates that the property (6), which enforces speed assignments with as close speeds as possible, does not always lead to the most energy-efficient solution. Nevertheless, the heuristic gives reasonable solutions for sufficiently large time intervals, as we will demonstrate in Sec. VI.

C. Combined Priority and Speed Assignment

Let us now consider the general case in which we want to determine both the priority assignment Π and the speed assignment Σ for a task set Γ. For this case we propose a heuristic that combines Alg. 3 and Alg. 2. More specifically, we integrate the priority reordering in the bottleneck speed assignment algorithm. At each loop iteration of the bottleneck algorithm, instead of just revising the global speed for a task sub-chain, we reorder the tasks in the sub-chain such that the minimal global execution speed is achieved for these tasks. In other words, we alternate task reordering and bottleneck finding in task sub-chains. This results in an algorithm with time complexity $O(N^4 \log \frac{1}{\epsilon})$. For the sake of conciseness, we omit the corresponding pseudo-code.

D. Adaptation of the Algorithms for Discrete Speeds

Practical processors typically provide a finite number of operating speeds/frequencies. In this subsection we illustrate how the above algorithms can be adapted to the discrete case in which $\mathcal{S} = \{s_1, s_2, \cdots, s_M\}$.

Alg. 1 and 2 can be directly reproduced for the discrete case. The only adaptation in Alg. 2 concerns the binary search over $\mathcal{S}$, which now is carried out over the discrete set of available speeds. The time complexity of this discrete version of Alg. 2 is $O(N^3 \log M)$, if M different execution speeds can be used. For Alg. 3 more care has to be taken with the reformulation for the discrete case, as there is no clear notion of bottleneck task anymore. After the assignment of a global discrete speed $\bar{s}$ to a sub-chain of tasks, we could still try to schedule these tasks with the next lower discrete speed and then identify the task τ^i_{pr} which results unschedulable. However, assigning $\bar{s}$ to all tasks τ^j_{pr} in the sub-chain with $j \leq i$ could be overly conservative, meaning that it could lead to a non-tight speed assignment. In particular, it could suffice to increase the speed level for just a few of those tasks in order to guarantee the schedulability of τ^i_{pr}. Hence, τ^i_{pr} cannot be considered the bottleneck for $\bar{s}$. A simple solution to this problem is to treat each priority level as bottleneck, that is, to repeat the speed assignment to lower priority tasks at each priority level. The corresponding pseudo-code is shown in Alg. 4. For the sake of readability, in Alg. 4 we use the notation $s(i)$ to refer to a discrete speed s_i. In order

Algorithm 4 `Determine discrete speeds`

```
1: function DETERMINE_DISCRETE_SPEEDS(Γ, Π, β, S): Σ
2:     if SCHED(Γ, Π, 1, n, s_M, β) = true then
3:         sl^0_pr ← M
4:         β^1_pr ← β
5:         for i ← 1, N do
6:             sl ← MINSPEEDLEVEL (Γ, Π, S, i, N, 1, sl^{i-1}_pr, β^i_pr)
7:             for k ← i, N do
8:                 sl^k_pr ← sl
9:                 β^{k+1}_pr = RT(β^k_pr, α^k_pr/s(sl))
10:            end for
11:            if sl = 1 then
12:                break
13:            end if
14:        end for
15:    else
16:        ERROR(UNSCHEDULABLE)
17:    end if
18:    return Σ = {τ^1_pr → s(sl^1_pr), ··· , τ^N_pr → s(sl^N_pr)}
19: end function

20: function MINSPEEDLEVEL(Γ, Π, S, From_pr, To_pr, sl_lo, sl_up, β_in): sl
21:     if SCHED(Γ, Π, From_pr, To_pr, s_1, β_in) = true then
22:         sl ← 1
23:         return sl
24:     end if
25:     repeat
26:         sl ← ⌈(sl_lo + sl_up)/2⌉;
27:         if SCHED(Γ, Π, From_pr, To_pr, s(sl), β_in) = true then
28:             sl_up ← sl
29:         else
30:             sl_lo ← sl
31:         end if
32:     until sl_up − sl_lo = 1
33:     return sl_up
34: end function
```

to find global speeds for sub-chains of tasks, the algorithm employs the function `MinSpeedLevel` which implements a binary search on the ordered set of discrete speeds. The time complexity of Alg. 4 is $O(N^2 \log M)$. Note that employing a brute-force algorithm for the discrete speed assignment would comport time complexity $O(M^N)$.

Finally, as done for the continuous case, we combine the algorithms for priority and speed assignment. In particular, each time we have to assign a global discrete speed to a task sub-chain, we reorder the tasks such that the minimum speed is obtained. Note that in the resulting heuristic we choose to reorder the tasks assuming *continuous* speeds, and then assign the next higher discrete speed to the task sub-chain. This is done because under discrete speeds the reordering procedure is less selective (assuming $\epsilon << |s_i - s_j| \ \forall s_i, s_j \in \mathcal{S}_{discr}$), meaning that it is more likely to return a priority order that restrains further speed reductions at lower priority levels. The described heuristic has time complexity $O(N^4 \log \frac{1}{\epsilon})$, versus $O(N! M^N)$ of a brute-force approach.

VI. Experimental Evaluation

In this section we evaluate the performance of the proposed algorithms. We compare the worst-case energy consumption of different systems when priorities and execution speeds for the single tasks are determined according to Algorithms 2, 3, 4, and the combined heuristics. We consider both continuous and discrete sets of execution speeds and compare the corresponding results. We also report the run-times of the different algorithms.

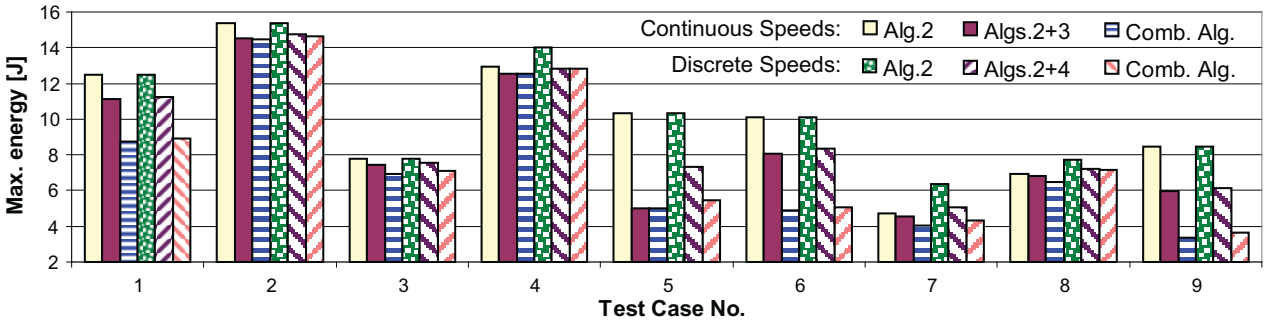

Fig. 6: Worst-case energy consumption in an interval of $\Delta = 10^4$ ms.

Experimental setup: We use the RTC Toolbox for Matlab[1] for our experiments. The RTC Toolbox provides data structures for an efficient handling of arrival and service curves and implements the described scheduling analysis. We construct nine test cases for the evaluation of the algorithms. Each test case consists of a processor that executes ten tasks according to a preemptive static priority scheduling policy. Each task is activated by an event stream represented by an arrival curve. In order to facilitate the replication of the experiments by means of other analysis tools, we use PJD-models (see Sec. IV-A) for the timing characterization of the event streams. For each test case we randomly generate integer parameters for ten tasks, as shown in Table I for test case 6. The periods p_i are chosen uniformly from the range $[5\text{ms}, 30\text{ms}]$, the jitters j_i from $[0, 2p_i]$, the minimum event inter-arrival times d_i from $[0, \lceil p_i/4 \rceil]$, and the execution times C_i from $[1\text{ms}, \lceil p_i/15 \rceil]$. We also associate relative deadlines D_i to the tasks.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}
p	14	28	15	30	28	7	25	25	30	22
j	20	35	3	19	35	5	30	20	0	15
d	2	2	0	0	4	0	5	0	0	0
C	1	1	1	1	1	1	2	1	1	2
D	21	1	53	406	276	41	6	342	4	181

TABLE I: Parameters for test case No. 6 [ms]

For each test case we automatically construct a modular scheduling analysis model and discard parameter sets that are not schedulable at the maximum speed. The arrival curves are obtained with $\alpha_i(\Delta) = C_i \cdot \bar{\alpha}_i(\Delta)$, where $\bar{\alpha}_i(\Delta)$ is determined according to Eq. 2. We let β represent a fully available processing resource and set the precision requirement for continuous speeds to $\epsilon = 10^{-4}$. The complete parameter sets for the nine test cases, the full list of the computed results, as well as the implementation of the algorithms are available online.[2]

For the evaluation we consider the Intel XScale platform with a maximum speed of 1GHz which we normalize to $s_{\max} = 1$. The approximated active power consumption function is then given by $\hbar(0.08 + 1.52s^3)$ Watt. In order to determine the worst-case energy consumption of the different systems under given priority and speed assignments, we implement a simple discrete-event simulator that reproduces system executions under maximum workload.

For each test case we consider two scenarios: A) continuous speed assignments, B) discrete speed assignments. For the two scenarios we assume $\hat{\mathcal{S}}_A = [0, 1]$ and $\hat{\mathcal{S}}_B = \{0.15, 0.4, 0.6, 0.8, 1\}$, respectively. After computing the critical speed we obtain $\mathcal{S}_A = [0.297, 1]$ and $\mathcal{S}_B = \{0.4, 0.6, 0.8, 1\}$. For each test case we consider three different priority and speed assignment policies and compare the corresponding worst-case energy consumption for a time interval $\Delta = 10^4$ ms. For scenario A the policies are as follows: (a) Compute Π and Σ_s with Alg. 2; (b) Compute Π' with Alg. 2, then compute Σ' with Alg. 3; (c) Compute Π'' and Σ'' with the combined heuristic. For scenario B the priority and speed assignment policies are defined accordingly.

Results: Due to space constraints we report the detailed priority and speed assignments for test case 6 only. Table II compares the priorities and speeds that result from the different assignment policies in the continuous scenario. The upper line in a row indicates which task has been assigned to the corresponding priority level. The lower line shows the execution speed assigned to the task. The table illustrates that a global speed assignment is in general pessimistic for a set of real-time tasks with static priorities, as the execution speeds of some tasks in the set might be reduced without harming timing constraints. In addition, the table shows that the combined optimization of priorities and speeds can lead to considerably better results than consecutive priority and speed assignments.

Policy \Priority	1	2	3	4	5	6	7	8	9	10
(a) Alg. 2	τ_2 1.000	τ_9 1.000	τ_7 1.000	τ_{10} 1.000	τ_8 1.000	τ_6 1.000	τ_1 1.000	τ_5 1.000	τ_4 1.000	τ_3 1.000
(b) Alg. 2 + 3	τ_2 1.000	τ_9 .9445	τ_7 .9445	τ_{10} .9445	τ_8 .9445	τ_6 .9445	τ_1 .9445	τ_5 .5776	τ_4 .5776	τ_3 .5776
(c) Combined	τ_2 1.000	τ_9 .7500	τ_7 .7500	τ_8 .5914	τ_1 .5914	τ_6 .5914	τ_3 .5914	τ_5 .5914	τ_4 .5914	τ_{10} .5914

TABLE II: Priority and speed assignments determined by the different policies for test case 6 and Scenario A

Fig. 6 sums up the the worst-case energy consumption deriving from all discussed assignment policies for all nine test cases. The chart shows that the results discussed for test case 6 apply also to the remaining test cases. In particular, the combined priority and speed assignment heuristic clearly outperforms the other algorithms. It achieves energy savings

[1] Available at `http://www.mpa.ethz.ch/Rtctoolbox`
[2] `http://www.tik.ee.ethz.ch/~psimon/FPDVS.zip`

up to 60% compared to the optimal priority ordering for minimum global speed (Alg. 2) and up to 44 % compared to the consecutive priority and speed assignment (Alg. 2 + 3). The results demonstrate that a large part of the optimization potential is sacrificed if priorities and speeds are not optimized in a combined manner. The chart of Fig. 6 also compares the worst-case energy consumption derived by the different algorithms under the discrete speed set $\mathcal{S}_B$ with respect to the continuous set $\mathcal{S}_A$. The comparison shows that in the discrete case more energy is consumed due to the coarser granularity of the speed assignments.

Scenario	Continuous (A)			Discrete (B)		
Assignment Policy	(a)	(b)	(c)	(a)	(b)	(c)
Min. run-time	3.4	4.5	7.8	0.6	1.2	15.6
Max. run-time	43.5	49.6	56.4	6.3	10.0	357.5

TABLE III: Min. and max. run-times measured for the different assignment policies [seconds].

In Table III we report the minimum and maximum run-times measured for the different algorithms. Note that for policy (b) the cumulative run-time of both employed algorithms is indicated. The table shows that for the considered test cases with ten tasks the algorithms compute the priority and speed assignments in less than one minute, and that in general the algorithms for the discrete setting are considerably faster than their continuous counterparts. This is simply because in the more coarse-grained discrete setting less solutions have to be explored. However, the combined heuristic is slower in the discrete case compared to the continuous case, with a maximum run-time of almost 6 minutes. This is because in both cases continuous speeds are assumed for the task reordering (cf. Sec. V-D), but in general Alg. 4 enforces more task reorderings than Alg. 3, as it reorders low-priority tasks at each single priority level.

VII. Conclusion

In this paper we presented a new methodology for the design of energy-efficient real-time event processing systems. We introduced different algorithms that statically determine the priorities and/or the execution speeds for multiple real-time tasks with non-deterministic release patterns. We considered both continuous and discrete execution speeds and commented on the time complexity of the different approaches. The priority assignment algorithm is based on the observation that rearranging higher-priority tasks does not affect the execution of lower-priority tasks. The main idea behind the speed assignment algorithm is to use priority-monotonic speeds in order to reduce the energy consumption. By combining the two approaches we devised a heuristic that assures energy-efficient task execution as well as observation of all real-time constraints. The presented algorithms are not bound to particular task release models, but support any arbitrary task release pattern that can be abstracted by means of arrival curves. We evaluated the performance of the algorithms by comparing the worst-case energy consumption achieved by the different approaches for several test cases. In future work will consider the extension of the discussed methods to the online case.

Acknowledgment

This work is funded by the Swiss National Science Foundation under grant number 200020-116594.

References

[1] N. C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical report, Dept. of Computer Science, Univ. of York.

[2] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *IEEE Real-Time Systems Symposium*, pages 95–105, 2001.

[3] X. Bin, Y. Yang, and S. Jin. Optimal fixed priority assignment with limited priority levels. In *5th Intl. Workshop on Advanced Parallel Programming Technologies (APPT)*, pages 194–203, 2003.

[4] K. Bletsas and N. Audsley. Optimal priority assignment in the presence of blocking. *Inf. Process. Lett.*, 99(3):83–86, 2006.

[5] J.-J. Chen, N. Stoimenov, and L. Thiele. Feasibility analysis of on-line dvs algorithms for scheduling arbitrary event streams. In *IEEE Real-Time Systems Symposium*, pages 261–270, 2009.

[6] R. Davis and A. Burns. Optimal priority assignment for aperiodic tasks with firm deadlines in fixed priority pre-emptive systems. *Inf. Process. Lett.*, 53(5):249–254, 1995.

[7] G. M. de A. Lima and A. Burns. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *IEEE Transactions on Computers*, 52:1332–1346, 2003.

[8] X. He and Y. Jia. Leakage-aware energy efficient scheduling for fixed-priority tasks with preemption thresholds. In *international conference on Advanced Data Mining and Applications*, pages 379–390, 2008.

[9] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis-the SymTA/S approach. *IEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.

[10] R. Jejurikar and R. K. Gupta. Procrastination scheduling in fixed priority real-time systems. In *CASES*, pages 57–66, 2004.

[11] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the Design Automation Conference*, pages 275–280, 2004.

[12] W.-C. Kwon and T. Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM Trans. Embed. Comput. Syst.*, 4(1):211–230, 2005.

[13] J. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, 2001.

[14] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *DAC*, pages 806–809. ACM, 2000.

[15] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[17] A. Maxiaguine, S. Chakraborty, and L. Thiele. DVS for buffer-constrained architectures with predictable qos-energy tradeoffs. In *CODES+ISSS*, pages 111–116, 2005.

[18] B. Mochocki, X. S. Hu, and G. Quan. Practical on-line dvs scheduling for fixed-priority real-time systems. In *RTAS*, pages 224–233, 2005.

[19] G. Quan and X. S. Hu. Energy efficient dvs schedule for fixed-priority real-time systems. *ACM Trans. Embed. Comput. Syst.*, 6(4):29, 2007.

[20] R. Racu, A. Hamann, R. Ernst, B. Mochocki, and X. S. Hu. Methods for power optimization in distributed embedded systems with real-time requirements. In *CASES*, pages 379–388, 2006.

[21] S. Saewong and R. R. Rajkumar. Practical voltage-scaling for fixed-priority rt-systems. In *RTAS*, page 106, 2003.

[22] D. Shin and J. Kim. Dynamic voltage scaling of mixed task sets in priority-driven systems. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(3):438–453, 2006.

[23] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. *ISCAS*, 4:101–104, 2000.

[24] F. Xie, M. Martonosi, and S. Malik. Intraprogram dynamic voltage scaling: Bounding opportunities with analytic modeling. *ACM Trans. Archit. Code Optim.*, 1(3):323–367, 2004.

[25] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 374–382. IEEE, 1995.

[26] H.-S. Yun and J. Kim. On energy-optimal voltage scheduling for fixed-priority hard real-time systems. *ACM Trans. Embed. Comput. Syst.*, 2(3):393–430, 2003.

[27] D. Zhu, R. G. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *ICCAD*, pages 35–40, 2004.