Online Energy-Efficient Hard Real-Time Scheduling for Component Oriented Systems

Da He  
University of Paderborn/C-LAB  
Paderborn, Germany  
Email: da.he@c-lab.de

Wolfgang Mueller  
University of Paderborn/C-LAB  
Paderborn, Germany  
Email: wolfgang@acm.org

Abstract—The energy efficiency becomes one of the most important concerns in mobile electronic systems design with mandatory requirements for low energy consumption, long battery life and low heat dissipation. Dynamic Power Management (DPM) and Dynamic Voltage and Frequency Scaling (DVFS or DVS) are two widely applied system level techniques to conserve system-wide power consumption. In the context of hard real-time systems, however, DPM and DVS have to be used with great caution in terms of timing constraints. In this article, we study the combined application of DPM and DVS for component oriented systems with hard real-time tasks and propose a simulated annealing based optimization algorithm and its online execution with constant complexity at each scheduling point. Additionally, our approach considers multiple low power states (sleep states) with non-negligible switching overhead. The experimental results show that our approach can achieve almost an optimal solution.

Keywords—Dynamic Power Management; Dynamic Voltage and Frequency Scaling; Hard Real-Time Systems; Online;

I. INTRODUCTION

Clearly, nowadays energy conservation has gained significant attention in the electronic systems design. Low power consumption, long battery life and low heat dissipation become mandatory development requirements and objectives to reduce the system operation costs. However, due to the rapid growth of system complexity and continuous advancement in deep submicron process technology towards nanoscale circuits, power management becomes more challenging. From the system level point of view, two widely accepted power reduction techniques, Dynamic Power Management (DPM) and Dynamic Voltage and Frequency Scaling (DVFS or DVS) can be applied to efficiently reduce the system-wide energy consumption, which is defined as the total energy consumption of all components in a component oriented system. Hereby, we mainly refer to hardware components, e.g. processor, memory or peripheral devices and we define a component as power manageable, if it can be managed by the DPM or DVS.

The main idea behind DPM is to shut down the component (switch to a sleep state) when it is idle and wake up it when required. If we ignore the power state switching overhead, a component can be switched to a sleep state whenever it is idle. However, during the state switching the retention of register contents and stabilization of power supply usually cause both energy and time costs. Therefore, [1] introduced the concept of break even time to capture this issue. The deeper the sleep state, the more power it saves, however, the longer break even time it requires. In this article we consider multiple sleep states with non-negligible state switching overhead. In contrast to DPM, the DVS technique is applied during the component is working and slows down the component by lowering the voltage. If only considering dynamic power consumption, the energy consumption in a time interval is a convex and increasing function of speed (frequency). However, due to the continuous advancement in deep sub-micron process technology towards nanoscale circuits, the leakage power becomes dominant, thus the energy consumption becomes merely a convex function. [2] defined the critical speed to cover this aspect, i.e. no task should ever run below this speed.

Both DPM and DVS try to save energy at the cost of performance. Therefore, they have to be used with great caution in hard real-time systems. In general, there exist multiple research addressing this problem. However, most of them are either concentrating only on the DPM strategy [3], [4] or only on DVS strategy [5], [6], [7], [8], [9]. Only some recent studies [10], [2], [11] have reported that if the system-wide energy consumption is considered, DPM and DVS are working contradictory. In particular, DPM strategy tries to run the task as soon as possible, so that more idle time can be utilized for sleeping, however, this needs to increase the processor speed and thus its energy consumption. In contrast, the DVS tries to slow down the components. However, this decreases the idle time and leads to reduced possibilities to shut down them. In fact, [12] has shown the $NP$-hardness of this problem. In the literatures, the system-wide energy minimization problem on component oriented real-time systems has not been sufficiently studied, especially with consideration of non-negligible state switching overhead. Our approach applies Simulated Annealing (SA) heuristic algorithm, which is fairly simple but quite efficient. Furthermore, speaking of DPM/DVS based energy-aware real-time scheduling, there are online approaches and offline approaches. Clearly, an online version is better suited in terms of flexibility, because it can be adaptive to the system changes. However, in the literatures, the existing online approaches are often only concentrating on dynamic slack, which comes from the task earlier completion before the worst case execution time, while the sophisticated techniques exploring static slack are only considered offline.
Our approach provides a unified online solution for both static slack and dynamic slack exploration. In addition, it comes with a computational complexity of $O(1)$ at each scheduling point, which provides significant advantages compared to other approaches that usually run at least in polynomial time. Nevertheless, even the logarithmic complexity is an obstacle for wide acceptance, as it is with $log(n)$ for EDF scheduling in practical applications.

In summary, our approach has two main contributions: i) the SA based algorithm and ii) its online execution. Since we are interested in the component oriented systems, the system-wide energy consumption is of utmost importance. We assume that the system is composed of a single processor with DPM and DVS capabilities and multiple peripheral devices with only DPM capability. In addition, our approach is not constrained to a fixed real-time schedule, both Earliest Deadline First (EDF) and Rate Monotonic (RM) can be applied.

The remainder of this article is organized as follows. Section II gives a brief outline of the state-of-the-art in the context of energy efficient scheduling in hard real-time systems. Section III introduces the system model based on Advanced Configuration & Power Interface (ACPI) and formally addresses the problem. In Section IV and V, we propose the simulated annealing based algorithm for online execution. Before Section VII concludes, the experimental results through simulation are presented in the Section VI.

II. RELATED WORK

There have been extensive research works in the field of applying DPM and DVS in real-time systems. Traditionally DPM and DVS are often considered disjointedly. In case of DVS approach the primary focus is on the power reduction of processors. One of the earliest optimal offline DVS algorithms with polynomial complexity is proposed by Yao et al. [5]. The work by Aydin et al. [6] provides a power aware DVS algorithm composed of both offline and online parts. In the offline phase the optimal frequency allocation is determined while in the online phase a dynamic slack reclaiming mechanism and a speculative speed adjustment are introduced to further reduce the processor power consumption. In [7] the authors proposed an online intra-task DVS algorithm in combination with EDF to exploit dynamic slack with computation complexity of $O(1)$. However, a continuous processor frequency range is considered. Kim et al. [8] compared various hard real-time DVS algorithms in terms of energy efficiency and performance under a unified framework. [9] introduced a full polynomial time approximation algorithm. However, none of the above mentioned work considered DPM with non-negligible state switching overhead. DPM related works are usually proposed for device scheduling. In [3] an online DPM algorithm in conjunction with EDF is proposed, where tasks are procrastinated as much as possible to create large device idle intervals. The work by Swaminathan et al. [4] proposed an offline optimal device scheduling algorithm for hard real-time systems based on pruning techniques. A heuristic search algorithm is proposed as well to find near optimal solution.

Only recently the relationship of DPM and DVS attracted more attentions in the context of system-wide energy efficient real-time scheduling. Generally, DPM and DVS are applied for devices and processors, respectively. Devadas et al. [10] studied the exact interplay of DPM and DVS. However, their focus is on frame based task model, where all tasks share common period/deadline. In the work of Jejurikar et al. [2] the concept of critical speed was introduced. The task should never run under the critical speed, otherwise the power consumption increases. However, they ignored DPM state switching overhead. Other related works are [12] and [13]. However, none of them is fully online. The authors of [12] proposed an approach composed of an offline and an online component. In [13] the proposed algorithm is only semi-online, because the system-wide optimal processor speed is computed offline. It can handle dynamic task changes but no device changes. Besides, [12] and [13] are tightly coupled with a fixed real-time schedule EDF. Niu [11] considered DPM and DVS capable processor and DPM capable devices. However, the proposed algorithm is not fully online either.

III. PRELIMINARIES

Before we propose the algorithm, the problem and the system model composed of a processor power model, a device power model and a task model are formally described in this section. We address the power model of the processor and devices following the ACPI recommendation, which is an open industry standard aiming at unifying the HW/SW interface for processor/device configuration and power management. ACPI allows the operating system directed power management and provides platform independent interface description. More specifically, we adopt the concept of C-states and P-states to describe the power states of the processor and D-states for device. The C-states describe different power states, which contain one active state and multiple low power (sleep) states with different sleep depth. The P-states reveal different performance states when the processor is running. Mainly they differ in operating frequency and power consumption. The D-states concept is similar to C-states, but applied on devices.

A. Processor Power Model

We denote a set of processor power states as $C_0, C_1, ..., C_e$, where $C_0$ is the only working state, i.e. the processor can only execute tasks in this state and $C_1, C_2, ..., C_e$ are low power states (sleep states) in non-increasing order of power consumption. These states are known as C-states in ACPI. $\forall i: 1 \leq i \leq e$ we define $P(C_i), L_{on\rightarrow off}(C_i), L_{off\rightarrow on}(C_i), P_{on\rightarrow off}(C_i)$ and $P_{off\rightarrow on}(C_i)$ as follows:

- $P(C_i)$ is the power consumption of the state $C_i$,
- $L_{on\rightarrow off}(C_i)$ is the latency for state switching from $C_0$ to $C_i$,
- $L_{off\rightarrow on}(C_i)$ is the latency for state switching from $C_i$ to $C_0$,
- $P_{on\rightarrow off}(C_i)$ is the power consumption for state switching from $C_0$ to $C_i$,
• $P_{off \rightarrow on}(C_i)$ is the power consumption for state switching from $C_i$ to $C_0$.

For a given processor power model we can easily compute the break even time for each low power state denoted as $BE(C_i)$ [1]. For the working state $C_0$ we further define power sub-states, which are known as performance states (P-states) in ACPI. We denote them as $S_1, S_2, ..., S_s$ where $S_1$ is the full performance state (with maximal voltage/frequency) and the remaining states are sorted in non-increasing order of power consumption. $\forall i: 1 \leq i \leq s$ we denote $F(S_i)$ and $P(S_i)$ as the corresponding frequency (normalized with regard to $S_1$) and the power consumption, respectively. The overhead for state switching among different P-states is accounted to the task worst case execution time.

B. Device Power Model

We denote a set of devices as $R_1, R_2, ..., R_m$. For each device we define the power states following the D-states concept in ACPI specification. We denote $D_{i,j}$ as the $j$-th power state of the device $R_i$. $D_{i,0}$ is the only active state and the remaining states are low power states (sleep states) in non-increasing order of power consumption. Furthermore, for the device power state $D_{i,0}$ we denote the power consumption as $P(D_{i,0})$. Similarly to the processor power model, for each low power state $D_{i,j}$ with $j > 0$ we define $P(D_{i,j})$, $L_{on \rightarrow off}(D_{i,j})$, $L_{off \rightarrow on}(D_{i,j})$, $P_{on \rightarrow off}(D_{i,j})$ and $P_{off \rightarrow on}(D_{i,j})$ as the power consumption and the switching overhead of the state $D_{i,j}$. The break even time for each low power state $D_{i,j}$ with $j > 0$ is denoted as $BE(D_{i,j})$. In the following text a component could be referred to either as a processor or a device.

C. Task Model

Since we are interested in hard real-time systems with periodic tasks, we adopted the classical real-time task model. The task set (independent tasks) is denoted as $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ with $W(\tau_i)$ denoting the Worst Case Execution Time (WCET) under maximal processor speed, $T(\tau_i)$ denoting the relative deadline (equal to the period) of the task and $Dev(\tau_i)$ denoting the set of components required by the task execution, i.e. $Dev(\tau_i) \subseteq \{\text{processor}\} \cup \{R_1, R_2, ..., R_m\}$. Note that the processor is required by all tasks. The hyper period is denoted as $HP$, which is the least common multiple of all task periods.

D. Problem Formulation

Before formulating the problem, we first introduce several definitions.

Definition 1. Given a system model as described previously, a configuration is defined as a mapping from the available P-states of the processor to the task set. It is denoted as $conf: \Gamma \rightarrow \{S_1, S_2, ..., S_s\}$.

Definition 2. Given a configuration and a real-time schedule (e.g. EDF or RM), the configuration is valid, if all tasks can meet their deadlines.

In this work, we assume that the task execution time is linear to the processor frequency. For a given configuration, the WCET of task $\tau_i$ is scaled up according to the assigned processor frequency. It can be easily computed as $\frac{W(\tau_i)}{W_{\text{max}}}$. In the case of RM we can then simply decide the validity of a configuration by means of testing whether the processor utilization is less than or equal to 0.693\(^1\), i.e. $\sum_{i=1}^{n} \frac{W(\tau_i)}{W_{\text{max}}} \leq 0.693$. In case of EDF the utilization upper bound is 1.

Definition 3. Given a configuration and a real-time schedule, the quality of a configuration is defined as the system-wide energy consumption over one hyper period. The less energy the system consumes, the better quality the configuration has.

Definition 4. Given a configuration and a real-time schedule, a power state schedule is defined as a schedule, which decides for all components, when they must be switched on to the active state and when can be switched off to a low power state and specifically to which low power state.

Due to the non-negligible power state switching overhead, a component must be switched on a little ahead of the dispatch time of the task, which requires the component. Otherwise, the task is delayed and the deadline may be jeopardized. On the contrary, by switching off components we must always ensure that the selected low power state is justified, i.e. the next idle interval must be longer than the corresponding break even time [1]. In general, deriving a power state schedule is a non-trivial job, because we need exact knowledge of the task start and finishing time.

Problem 1. Given a system model and a real-time schedule as the input, the output is to find the optimal configuration, which is valid and has the best quality. Additionally, a power state schedule should be derived.

Unfortunately, the above described problem is \(NP\)-Hard, because the authors in [14] have shown that even ignoring the power state switching overhead, the problem is already \(NP\)-Hard.

IV. SIMULATED ANNEALING BASED ALGORITHM

Since the problem is \(NP\)-hard, we decided to apply simulated annealing algorithm to solve the problem. The main idea of SA is to iteratively improve the solution by investigating the neighbours. If a neighbour solution is better, a movement to the neighbour solution is made, otherwise the movement is only made with a certain probability. The process repeats until a solution with certain quality is found or the iteration number reaches a predefined threshold. One of the advantages of SA is being able to escape from a local optimum, which is a solution with all its neighbour solutions being worse that itself.

Before we explain the algorithm in detail, we first define the neighbourhood of a configuration in our context. Two

\(^1\)The least upper bound of processor utilization for RM is $n(2^{1/n} - 1)$ and converges to 0.693 for high values of $n$. 
configurations are neighbours, if they exactly differ in the frequency assignment of one task. Furthermore, we take the algorithm CS–DVS from [2] to compute our initial configuration. Algorithm 1 illustrates our main algorithm.

**Algorithm 1 Simulated Annealing**

**Require:** The system model

**Ensure:** A configuration

1: Generate initial configuration according to CS–DVS
2: Evaluate quality of the initial configuration
3: while iteration < length do
4: Randomly generate a neighbour configuration as the new configuration
5: if the new configuration is valid then
6: Evaluate the new configuration and accept the new configuration with probability $p$
7: end if
8: iteration = iteration + 1
9: end while

In this section we assume that the quality of a configuration can be obtained in some way and the details are explained later in the subsequent section. In Algorithm 1 we have two parameters length and $p$. The length specifies the total iteration number of the algorithm. The $p$ is the acceptance probability and is defined as in Equation 1, where $K$ is a constant, $q$ and $q'$ are the quality of the current configuration and the new configuration, respectively.

$$p = \begin{cases} 1, & \text{if } q > q' \\ \exp\left(\frac{q-q'}{T+K}\right), & \text{otherwise} \end{cases} \tag{1}$$

V. ONLINE EXECUTION

Traditionally, the SA is usually implemented in an offline fashion, since all required information is available before runtime. However, several issues will arise when we try to implement the Algorithm 1 offline:

- How to obtain the quality of a configuration? If we ignore the state switching overhead, then a component can be switched to a low power state whenever it is idle. Thus the energy consumption of a component can be mainly computed by multiplying the power consumption of the component with its active time. However, in case of non-negligible state switching overhead, the calculation of energy consumption becomes much more difficult, because we need to analyse the exact length of each idle interval, based on which we are able to decide whether the component can be switched to a low power state or not. The idle interval analysis requires the exact knowledge of the start, preemption, resume and finishing time of each task. As a consequence, this analysis has similar computation complexity as the response time analysis.

- How to derive the power state schedule? The Algorithm 1 actually does not answer this question directly. A power state schedule has to be derived according to the configuration found by Algorithm 1. If Algorithm 1 is implemented offline, then this schedule is to be derived offline as well. This requires similar computation complexity as the response time analysis as well.

- How to deal with dynamic slack? Since the dynamic slack is not available offline, the offline algorithm can only explore static slack.

- How to handle dynamic system changes? The offline algorithm lacks the ability to be adaptive to the system changes, e.g. a new task is added into the system during runtime or a new device is plugged into the system.

In this article we solved the above mentioned problems by proposing an approach that allows the Algorithm 1 to be executed in an online and adaptive fashion. As will be explained later, in our online approach the evaluation of the configuration quality and deriving of the power state schedule will become trivial jobs. Hereby the main challenge is to integrate the SA algorithm into the hard real-time system without missing any deadlines. The basic idea is to take the advantage of the common feature of both domains, namely “iterative”. The SA algorithm is iterative, because it iteratively improves the candidate solutions, while the real-time system with periodic tasks is also iterative in terms of the iterative task execution in each hyper period. In our approach each iteration of the SA algorithms is mapped to a hyper period. With other words, in each hyper period we explore and evaluate a candidate configuration. This stage is called Exploration Stage (ES) and is stopped after a certain number of hyper periods. For the remaining time we call it the Application Stage (AS), because in this stage we apply the best configuration we have found in the exploration stage.

Moreover, we assume that all tasks run until their WCET during ES. If a task finishes earlier than its WCET, then we artificially prolong its execution time to the WCET. In other words, we explore only static slack in the ES and the dynamic slack is explored during AS. Another important note is that, since we have no knowledge of the power state schedule at the beginning, we keep all components always active during the entire ES, so that no task will be delayed due to the state switching overhead. However, in the AS we will derive a power state schedule.

A. Exploration Stage

In this stage, the main goal is trying to find the best configuration. Mainly we will explore one configuration in one hyper period. During runtime we perform two activities to achieve this goal.

The first activity with the name Algorithm Activity (AA) occurs at the end of each hyper period. The AA mainly performs the work specified for each iteration in Algorithm 1. First the quality of the current configuration has to be evaluated. Since we perform runtime energy recording, which will be explained later, the quality of the configuration can be easily obtained. Afterwards the acquired quality is to be compared with the quality of the configuration from the previous hyper period. According to the acceptance probability
a movement to the current configuration is made. Finally the next configuration will be generated, which is a neighbour configuration of the current configuration. Before the next hyper period starts, a schedulability test is carried out to test if the new configuration is valid. In case of invalidity the current configuration is used in the next hyper period.

The second activity with the name **Recording Activity (RA)** happens at each scheduling point. In RA we mainly record the component activation/deactivation events and the energy consumption. The events for each component are collected in an event list over each hyper period. Each event contains two information: i) the time stamp when it is recorded, ii) if it is an activation event or deactivation event. Fig. 1(a) shows an example illustrating the events $e_1$, $e_2$, $e_3$ and $e_4$ recorded into the event list of the device $R_1$ and $e_5$, $e_6$, $e_7$, $e_8$, $e_9$ and $e_{10}$ into the event list of the processor. We observe that all events of $R_1$ are related to the execution of the task $\tau_1$, this is because $R_1 \in Dev(\tau_1)$. Note that in Fig. 1(a) the $R_1$ is kept always on as mentioned earlier, even when it is not needed (between 20ms and 40ms). The recorded event list reflects the component behaviour, when it should be activated and when it could be deactivated. These event lists will be used in the AS to derive the power state schedule. Another note is that the event list of the processor actually collects the activation/deactivation events of all tasks. The required memory for storing the events is clearly dependent on the length of a hyper period. Theoretically the hyper period of a task set could be arbitrarily long, however, in most real life applications the tasks are usually harmonic and therefore the hyper period is manageable.

The energy recording is very simple as well. Since the RA takes place during runtime, we know exactly when a task starts and finishes. At each scheduling point we first compute the energy consumption of involved components in the time interval between the previous recorded event and the current event, then this is cumulatively added to the total energy consumption of current hyper period. At the end of a hyper period we will automatically obtain the total energy consumption, which is the quality of the configuration. In Fig. 1(a), for instance, at the scheduling point 20ms the task $\tau_1$ finishes. The involved components are the processor and $R_1$, because they are in $Dev(\tau_1)$. Here we discuss the computation of the energy consumption for $R_1$ during the time interval from the previous event $e_1$ to the current event $e_2$. In this interval we notice that the $R_1$ should be active, because $\tau_1$ was active and therefore the energy consumption is computed by means of $P(D_{1,0}) \cdot (t_2 - t_1)$. If we assume that $t_1$ denotes the time stamp of $e_i$. At the scheduling point 40ms, we notice that the $R_1$ was idle during the time interval from the previous event $e_2$ to the current event $e_3$. The idle length is then $l = t_3 - t_2$ and the energy consumption $E$ is computed via Equation 2. If a component supports multiple low power states, we select the state with the largest break even time that is less than the length of the idle interval. In this example we assume that the $R_1$ supports only one lower power state. Note that here the scheduling point 30ms is not relevant for $R_1$, because at that scheduling point the $R_1$ is not involved.

$$E = \begin{cases} E_{on \rightarrow off} + E_{off} + E_{off \rightarrow on} & \text{if } l > BE(D_{1,1}) \\ P(D_{1,0}) \cdot l, & \text{otherwise} \end{cases}$$

(2)

where $E_{on \rightarrow off}$ is the energy consumption of switching off the component, $E_{off \rightarrow on}$ is the energy consumption of switching on the component and the $E_{off}$ is the energy consumption of the component when it is off. They can be expressed by Equations 3. Note that the energy recording for each component needs only $O(1)$ operation at each event.

$$E_{on \rightarrow off} = P_{on \rightarrow off}(D_{1,1}) \cdot L_{off \rightarrow on}(D_{1,1})$$

$$E_{off \rightarrow on} = P_{off \rightarrow on}(D_{1,1}) \cdot L_{off \rightarrow on}(D_{1,1})$$

$$E_{off} = P(D_{1,1}) \cdot (l - L_{on \rightarrow off}(D_{1,1})) - L_{off \rightarrow on}(D_{1,1})$$

(3)

In summary, the recorded event lists reflect the time behaviour of the components, when they should be activated and when they could be deactivated. The recorded energy is the energy consumed by the components, if we schedule the components according to the recorded event list.

**B. Application Stage**

In this stage we apply the best configuration found in the ES and derive the power state schedule from the recorded events. According to the Definition 4 the power state schedule has the work to decide when a component is to be switched off and when to be switched on. For this purpose we perform an activity called **DPM Activity** at each scheduling point. If there

---

Fig. 1. One Example with $Dev(\tau_1) = \{\text{processor, } R_1\}$ and $Dev(\tau_2) = \{\text{processor}\}$
is any task finishing or being preempted at a scheduling point, then the components required by the task can be potentially switched to a low power state, however, this decision is dependent on the length of the upcoming idle interval. Other existing online approaches are often applying complicated techniques for estimating the next activation time (even with task procrastination), which usually take polynomial time at least. On the contrary, we can simply consult the event list to compute the length of the next idle interval, which is much more efficient. According to this information we can decide whether we should switch a component to a particular low power state. If a component is switched to a low power state, we also need to switch on it a little ahead of the next actual required time. Fig. 1(b) illustrates this DPM activity without consideration of any dynamic slack. Here we assume that we use the configuration and event list from Fig. 1(a). Since no task runtime variation is considered, the task execution behaves exactly the same as in Fig. 1(a). At the scheduling point 20\text{ms}, the task \( \tau_1 \) finishes and the processor and \( R_1 \) can be potentially put into low power states. After consulting the event list of the processor and \( R_1 \), respectively, we know that the processor will be required by 30\text{ms} and \( R_1 \) will be required by 40\text{ms}. For example, both idle intervals are large enough, then both processor and \( R_1 \) are switched to a proper low power state. If a component supports multiple low power states, we select the state with the largest break even time that is less than the length of the idle interval. Note that the components are switched on a little ahead of their actual required time. The switching on/off process is illustrated by triangles in Fig. 1(b) and Fig. 1(c).

As mentioned before we also would like to explore the dynamic slack in AS. Here we adopted the dynamic slack reclaiming mechanism. More specifically, we utilize the unused execution time of a task for additional power saving via DPM. Mainly the components can be switched off earlier than the worst case. Fig. 1(c) shows the same example as in Fig. 1(b), but all tasks are finishing earlier than their WCET. At the scheduling point where the task \( \tau_1 \) finishes (at 14\text{ms}), we can consult the event list of the processor and \( R_1 \), respectively. As a consequence, we know that the processor will be required by 30\text{ms} and \( R_1 \) will be required by 40\text{ms}. This information will guide us to compute the length of the upcoming idle interval, which can be used to make the decision whether a component should be switched to a low power state. One important note here is that all tasks are always activated at the recorded activation time. If a task is ready earlier than the recorded time due to the earlier completion of the previous task, then it needs to be delayed to the recorded activation time, because only at that time we can guarantee that all the required components are in active state. The second instance of task \( \tau_1 \) in Fig. 1(c) illustrates this situation. It becomes ready after the task \( \tau_2 \) completes (at 34\text{ms}), however, it should wait until its recorded activation time, which is at 40\text{ms}. More details can be seen in Algorithm 2 in the APPENDIX.

Generally, our approach is launched at the system start and runs from ES to AS. As soon as there are any system changes, such as new tasks are added into the system or a new device is plugged into the system (only allowed at hyper period boundaries), the approach will start over with the calculation of the new initial configuration by means of \textit{CS-DVS} algorithm and run from ES to AS again.

\textbf{C. Correctness and Complexity}

We prove the correctness and the efficiency of our approach through two theorems.

\textbf{Theorem 1.} Our online approach can always guarantee the system schedulability.

\textbf{Proof:} Since our approach is divided into two stages, we show the system schedulability in ES and AS, respectively. In ES we explore one configuration in one hyper period. Since the initial configuration obtained by means of \textit{CS-DVS} is clearly valid, in the first hyper period there will be no task missing its deadline. Note that in ES all components are never switched off (even when they are idle), thus no task will be delayed by state switching overhead. This is also the reason why the classical schedulability test via utilization can be performed. At the end of the first hyper period the configuration for the next hyper period is generated. As a schedulability test is performed, and only if the test is positive, the generated configuration will be used in the next hyper period (otherwise the initial configuration is used), thus the schedulability is guaranteed as well. The procedure repeats at the end of each hyper period, therefore the configuration used in each hyper period is obviously valid.

In AS the best found configuration is applied and obviously this configuration is valid. Furthermore, since all components are switched on as recorded and all tasks are activated as recorded, no task will be delayed and therefore no task will finish later than its recorded finishing time, which is obtained in the worst case scenario. Therefore no task will miss its deadline. In total the system schedulability is always guaranteed.

\textbf{Theorem 2.} Our online approach has \( O(1) \) complexity at each scheduling point.

\textbf{Proof:} In ES we mainly perform two activities, the algorithm activity and the recording activity. In the algorithm activity we first evaluate the current configuration. The evaluation takes \( O(1) \), because through the runtime energy recording the energy consumption of a configuration is automatically available at the end of a hyper period. The generation of a neighbour configuration takes \( O(1) \) as well, because we only need to change the frequency assignment of one task. As a consequence, the processor utilization (for schedulability test) can also be obtained by updating the utilization of the changed task. Therefore the algorithm activity takes \( O(1) \) time. In the recording activity we record the events and energy for each involved components at each scheduling point. Obviously, the events and energy recording for a single component takes only \( O(1) \). At one scheduling point there can be at most \( m \) components getting involved, where \( m \) is the number of
components in the system. Therefore the recording activity takes $O(m)$ time at each scheduling point.

In AS the most time consuming work is to find the next activation time for each involved component at the scheduling point. Since the event list contains the events in non-decreasing order of their time stamps. We could find the proper event in time $O(\log(L))$ via binary search, where $L$ is the length of the event list. However, if we manage an index variable addressing the elements in the list and advance it properly at each scheduling point, then each time we only need to retrieve the next event according to the current index, which obviously takes only $O(1)$. More details can be seen in Algorithm 2 in the APPENDIX. Since the number of involved components is limited by $m$, the DPM activity takes $O(m)$ as well.

If we assume that the number of components in the system is relatively small and constant, we have the total runtime of $O(1)$ at each scheduling point. $lacksquare$

VI. Evaluation

We evaluated our concept by implementing an experiment setup using SystemC, which is an event-driven simulator. In the experiment we selected Intel XScale processor model from [14]. The device model is taken from [3] and five devices are used: MaxStream Wireless Module, IBM Microdrive, SST Flash, SimpleTech Flash Card and Realtek Ethernet Chip. All devices support one active state and one low power state. We performed both synthetic and realistic task model in the experiments.

A. Synthetic Task Model

In this experiment we randomly generated 500 task sets and the size of each task set is between 3 and 9. The period of each task is within $[0.05ms, 100ms]$. The WCET of each task is also randomly generated, so that the utilizations of the task sets are uniformly distributed in the interval $(0, 1)$. Furthermore, for each task we randomly select $0 \sim 2$ devices into the required device set. The EDF is applied as the real-time schedule (RM can also be applied) and the constant $K$ in Algorithm 1 is selected to be 0.001 according to our preliminary experiments. Hereby, we simulated some samples from the generated task sets with different $K$-values, where $K = 0.001$ delivers the best average result.

Fig. 2 shows the simulation result in terms of the number of tasks in the system (x-axis). The y-axis denotes the system-wide energy consumption (the quality of a configuration), which is normalized with regard to the configuration obtained by CS-DVS. In the legend the SA20hps, SA50hps and SA200hps indicate the configuration obtained by our SA algorithm after 20, 50 and 200 hyper periods in ES, respectively. The Optimum is the optimal configuration, which is obtained by investigating all possible configurations in simulation. This is not practical in the reality, however, it gives us a lower bound of the result for evaluation purpose. As observed, our algorithm achieves more power saving, if we spend more hyper periods in ES. Besides, the configuration obtained by SA200hps is already very close to the optimum.

![Fig. 2. The impact of task number](image)

![Fig. 3. The impact of utilization](image)

Fig. 3 shows the simulation result in terms of the utilization of the task set. As expected, if the system is heavily loaded (with high utilization), then there is little static slack. As a consequence, the system-wide energy consumption can be barely reduced. On the contrary, the power consumption can be reduced by factor of ca. 13%, if the utilization is relatively small.

Since our approach supports dynamic slack exploration, we performed an experiment with the task runtime variation, which is simulated according to Gaussian distribution. We denote $W_i$ and $B_i$ as the worst case and best case execution time of the task $\tau_i$, respectively, and $B_i$ is defined as $B_i = W_i \ast (1 - \alpha)$ with $0 < \alpha \leq 1$. Our Gaussian function is then defined with dependence to the task worst case execution time $W_i$ and the parameter $\alpha$, namely the mean is $\mu = \frac{W_i + B_i}{2}$, and the variance is $\sigma^2 = 0.2 \ast (W_i - B_i) = 0.2 \ast \alpha \ast W_i$. With other words, the greater the $\alpha$, the more variation is allowed. The Fig. 4 shows the simulation results. As expected, the greater the $\alpha$, the more dynamic slack can be utilized and therefore the more power can be saved.

B. Realistic Task Model

In order to make the evaluation more realistic, we performed 4 real life case studies taken from [15]. TABLE I shows the simulation results. The power reduction is acquired by comparing the configuration found after 200 hyper periods in ES with the CS-DVS configuration.
VII. CONCLUSION

This article introduced a fully online approach based on simulated annealing algorithm for energy efficient scheduling in component oriented hard real-time system. The computation complexity is $O(1)$ at each scheduling point. Our approach is independent of real-time schedules. Both EDF and RM can be used. Furthermore, our approach considers multiple low power states with non-negligible state switching overhead for DPM strategy. Both static slack and dynamic slack can be explored in an online fashion. The experimental results show the simplicity of our approach but yet efficient power reduction of ca. 13% in comparison with CS-DVS approach. The experiments obtained an almost optimal configuration already after 200 hyper periods.

VIII. ACKNOWLEDGEMENT

This work was partly funded by the DFG Collaborative Research Centre 614 and by the German Ministry of Education and Research (BMBF) through the BMBF project SANITAS (01M3088) and the ITEA2 projects VERDE (01IS09012), and TIMMO-2-USE (01IS10034A).

REFERENCES


APPENDIX

The variable $comp.index$ denotes the current index in the event list of the component $comp$ and is initialized with 1.

Algorithm 2 DPM Activity at each scheduling point in AS

1: if There is a task $\tau_i$ finishing or being preempted then
2: for all $comp \in Dev(\tau_i)$ do
3: Retrieve the next activation event according to $comp.index$ and compute the length of the next idle interval. Based on the length we decide, whether we can switch $comp$ to a proper lower power states. If $comp$ is switched off, we will wake up it a little ahead of the next activation time. Advance the index variable $comp.index$
4: end for
5: end if
6: if There is a task $\tau_i$ starting or being resumed then
7: if Current time is equal to the recorded time then
8: Run the task $\tau_i$
9: for all $comp \in Dev(\tau_i)$ do
10: Advance the index variable $comp.index$
11: end for
12: else
13: Wait until the recorded time
14: end if
15: end if

TABLE I
SIMULATION RESULTS OF REAL CASE STUDIES

<table>
<thead>
<tr>
<th>Case study</th>
<th>Task number</th>
<th>Power reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear Motor Control</td>
<td>3</td>
<td>27%</td>
</tr>
<tr>
<td>CNC Machine (without GUI)</td>
<td>7</td>
<td>10%</td>
</tr>
<tr>
<td>Airbag Control</td>
<td>9</td>
<td>18%</td>
</tr>
<tr>
<td>Electronic Stability Control</td>
<td>7</td>
<td>10%</td>
</tr>
</tbody>
</table>

Fig. 4. Simulation results with runtime variation