A Heuristic Energy-Aware Approach for Hard Real-Time Systems on Multi-Core Platforms

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—Nowadays, Dynamic Power Management and Dynamic Voltage (and Frequency) Scaling are well accepted for adjusting the trade-off between the performance and power dissipation. In this article, we investigate the problem of combined application of DPM and DVS in the context of hard real-time systems on cluster-based multi-core processor platforms. We propose a heuristic algorithm based on simulated annealing and its online execution. Our approach considers multiple low power states with non-negligible state switching overhead. The experimental results show that our algorithm can significantly reduce the power consumption in comparison with existing algorithms.

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

I. INTRODUCTION

Dynamic Power Management (DPM) and Dynamic Voltage (and Frequency) Scaling (DVS) are two well established system-level techniques to obtain the best trade-off between the system performance and power consumption during runtime. In general, the main idea behind DPM is to shut down the unused system components and wake them up when required. Since the switching on/off process usually needs to retain the register contents and stabilize the power supply, a mindless ignorance is not always justified. For instance, the entry latency of deep-sleep mode on Intel® PXA270 takes 600µs [1] and is obviously non-negligible if the task execution time is in the same order of magnitude. Thus, Benini et al. [2] introduced the concept of break even time to capture this issue. In contrast to DPM, the DVS technique is applied during the components are in active state and tries to slow down them to achieve power saving. If only dynamic power consumption is considered, the energy consumption of a component in a time interval is a convex and increasing function of speed (frequency). 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. The critical speed is defined to cover this aspect [3], i.e. no tasks should ever run below this speed. Note that we are not interested in scaling down the frequency while keeping the supply voltage, because it is not beneficial from the energy saving point of view.

In general, both DPM and DVS have to be used with great caution in the context of hard real-time systems due to task deadlines. In fact, the problem of the optimal application of DPM and DVS for hard real-time tasks is \( NP \)-hard [4]. Additionally, in terms of multi-core platforms the DPM and DVS can be applied in different levels, either on the whole processor chip or on the individual cores. They are referred to as full-chip platforms and per-core platforms, respectively. In the early years, the full-chip platforms are most common, because only one power supply net is required for the entire chip, which is obviously cost-efficient. However, they lack the flexibility for power management, because all cores can only operate at the same speed. With the introduction of Frequency/Voltage Island and on-chip voltage regulator, per-core platforms gain more and more interests. However, it suffers the problem of high implementation costs, which could become impractical if the number of cores dramatically increases. In this article our focus is on the cluster-based multi-core platform, which is a general form of per-core and full-chip platform and offers the best compromise. The cores are divided into clusters and the cores in the same cluster must operate at the same frequency. Furthermore, we concentrate on the partitioned scheduling, which provides the advantage that the well-established uniprocessor real-time scheduling, such as Earliest Deadline First (EDF) and Rate Monotonic (RM), can be adopted.

In this work we propose a simulated annealing based heuristic algorithm to minimize the energy consumption of hard real-time systems on cluster-based multi-core platforms, which are rarely addressed in existing research work. Besides, our algorithm considers multiple sleep states with non-negligible state switching overhead, which is often ignored in existing studies as well. Furthermore, in the context of DPM/DVS based energy-aware real-time scheduling there are online and offline approaches. Obviously, the online approaches are more advanced in terms of the flexibility, since they are adaptive to the system changes. However, in the literatures, most of the online approaches consider only dynamic slack and lack the ability to explore the static slack, because the sophisticated static slack exploration algorithms are usually very time-consuming. In this work, our another contribution is to propose a technique allowing our algorithm to be executed in an online fashion, which enables a completely online solution to explore the static and dynamic slack with logarithmic time overhead at each scheduling point.

The remainder of this article is organized as follows. Section
II gives an overview of related work. In section III we define the system model and the problem. Section IV describes the details of the main algorithm and section V shows how it could be performed in an online fashion with logarithmic complexity. Finally, before we conclude the work in section VII, the experiment results are presented in section VI.

II. RELATED WORK

In the context of multi-core processor platforms, the problem of energy efficient real-time scheduling has been discussed in several studies. Aydin et al. [4] showed the $N/P$-hardness of the problem of energy-aware task partitioning and proposed a load balancing framework based on per-core platforms. Chen et al. [5] described a per-core DVS algorithm taking different power characteristics of tasks into consideration, however, they assumed the frame-based task model and ideal processor, where all tasks have a common deadline and the cores can operate at any frequency within a given range. With regard to the leakage power the same authors provided several approximation algorithms in another work [3] on the per-core platforms, but the analyses are based on the ideal processor model as well. Lee et al. [6] classified the tasks into three categories according to their utilization and proposed a heuristic scheduling scheme based on the classification. In the domain of full-chip platforms, most work are concentrating on balanced task partition. Seo et al. [7] introduced the dynamic repartitioning algorithm based on existing partition, which tries to balance the task load on different cores by considering dynamic slack. A dynamic core scaling scheme was proposed as well to reduce the number of unused cores. Yang et al. [8] proposed an approximation algorithm to schedule a set of frame-based tasks.

Only recently, more and more attentions have been put on the cluster-based multi-core platforms. Kolpe et al. [9] have focused on the problem of clustering the cores into DVS domains. The authors proposed to group similar cores into cluster according to their typical workload. Chakraborty et al. [10] introduced a fundamentally alternate means for cluster-based multi-core processor design. They believe that a core, which is designed for a dedicated frequency/voltage domain, is more energy efficient than a core designed with runtime DVS capabilities and configured to that frequency/voltage domain. Both work are not focusing on hard real-time systems. The closest related work are presented by Kong et al. [11] and Qi et al. [12]. Both of them considered the cluster-based multi-core platforms and hard real-time systems, however, they both assumed frame-based task model and ideal processor model. In this work, we will address a more general task model and non-ideal processor model, which are more common in the reality.

III. PRELIMINARIES

In this section we introduce the system model and the problem. The system model is composed of a processor power model and a task model.

A. Processor Power Model

We specify the processor power model following the Advanced Configuration & Power Interface (ACPI) recommendations. More specifically, we adopt the concept of C-states and P-states to describe the power states of individual processor cores. The C-states describe the different power states containing one active state and multiple low power (sleep) states with different sleep depth. The P-states reveal different performance states when the processor core is active and mainly differ in the operating frequency and power consumption.

The power model of a cluster-based multi-core platform is defined as follows. The cores in the same cluster share the same power model and the cores from different clusters could possess different power models. In other words, the cores in the same cluster must operate at the same frequency. If all cores in the cluster are idle, the cluster can be switched to one low power state. Note that all cores in the cluster are put into the same low power state. For each core $D_{x,y}$, where $x$ is the cluster index and $y$ is the core index, we denote a set of power states (C-states) as $C_0, C_1, ..., C_c$, where $C_0$ is the only active state and $C_1, ..., C_c$ are low power states in non-increasing order of power consumption. Furthermore, $\forall i: 1 \leq i \leq c$ we define $P(C_i)$, $L_{on \rightarrow off}(C_i)$, $L_{off \rightarrow on}(C_i)$, $P_{on \rightarrow off}(C_i)$, $P_{off \rightarrow on}(C_i)$ and $T_{be}(C_i)$ as follows:

- $P(C_i)$ is the power consumption of the state $C_i$
- $L_{on \rightarrow off}(C_i)$ is the latency of the state switching from $C_0$ to $C_i$
- $L_{off \rightarrow on}(C_i)$ is the latency of the state switching from $C_i$ to $C_0$
- $P_{on \rightarrow off}(C_i)$ is the power consumption of the state switching from $C_0$ to $C_i$
- $P_{off \rightarrow on}(C_i)$ is the power consumption of the state switching from $C_i$ to $C_0$
- $T_{be}(C_i)$ is the break even time for the state $C_i$

The active state $C_0$ contains a set of P-states denoted by $S = \{S_1, S_2, ..., S_n\}$, where $S_1$ is the full performance state and the remaining states are in non-increasing order of operating frequency. $\forall i: 1 \leq i \leq s$ we denote $F(S_i)$ and $P(S_i)$ as the frequency and power consumption of the state $S_i$, respectively. Without loss of generality, the frequency is normalized with regard to the maximal frequency. Furthermore, the critical speed is denoted as $S_{cs}$ and is the most power efficient P-state, i.e. with the smallest $P(S_i) / F(S_i)$ (watt per hertz).

B. Real-Time Task Model

Since we are interested in hard real-time systems with independent periodic tasks, we adopt the classic real-time task model. The task set is denoted by $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}$ with $\forall i: 1 \leq i \leq n, \tau_i = (W_i, T_i)$ where $W_i$ denotes the Worst Case Execution Time (WCET) at maximal speed and $T_i$ is the relative deadline (equal to period) of the task. The hyper period of the task set is the least common multiply of all task periods. Furthermore, we assume that the task WCET increases linearly when the processor speed decreases.
C. Problem Definition

The problem of power optimization of hard real-time systems on multi-core platforms is composed of three parts: the task partitioning (processor affinity), the frequency assignment to each task and the task scheduling. In order to find the optimal solution, all parts should be considered coherently, because they have influence on each other. In this article, we concentrate on the first two parts and assume that the last part is solved by a priori given real-time schedule like EDF or RM. The main advantage is that our approach can be built on top of those well-established schedules. Thus, our problem is stated as follows.

Problem 1. Given a system model (including a multi-core processor power model and a task model) and real-time schedules on each core, the output is to find a solution including a task partition and a frequency assignment, so that all tasks are schedulable and the system energy consumption over one hyper period is minimal.

Formally, the solution is expressed by two functions alloc : \( \Gamma \rightarrow D \) and assign : \( \Gamma \rightarrow S \), where \( S \) and \( D \) are the set of P-states and processor cores, respectively. Furthermore, we refer to the system energy consumption over one hyper period as the solution value.

IV. SA based Heuristic Algorithm

In this section we propose a Simulated Annealing (SA) based heuristic algorithm to solve the Problem 1. The SA is inspired by the annealing process in material science and designed generally for solving combinatorial optimization problems. The main idea behind SA is to iteratively improve the solution by investigating the neighbour solutions. If a neighbour solution is better, a movement to the neighbour solution is made, otherwise the movement is accepted with certain probability. The process repeats until a solution with certain quality is obtained or the number of iterations reaches a predefined threshold. The SA is able to escape from local optima by allowing acceptance of worse solutions. In fact, Mitra et al. [13] has proved that the SA will eventually converge to a global optimum, if the iteration number is sufficiently large. In general, the SA algorithm is problem-independent and applicable for a large variety of problems. Usually, the neighbours are randomly selected by uniform probability. Even though in many cases the generalized SA already produces excellent results, we believe that there is still great improvement potential by adapting the generalized SA to a particular problem. Our idea is to apply problem specific and heuristic information to guide the selection of particular neighbours, so that, hopefully, the SA can converge to an optimal solution more rapidly. In our context, two solutions are in a neighbourhood, if they differ in the configuration of exact one task, i.e. the core allocation and the frequency assignment. A neighbour solution is generated in three steps: i) select a task ii) reallocate it to a new core and iii) reassign it with a new frequency.

For the task selection (the first step) we define the guidance based on the heuristic information learned from the current solution. More precisely, each task \( \tau_i \) is associated with a penalty value denoted by \(\text{pen}(\tau_i)\). Intuitively, the penalty value of a task indicates the wasted energy by the task. The higher the penalty value, the more possible the corresponding task is to be selected for configuration change. The \(\text{pen}(\tau_i)\) is formally computed by means of the Equation 1, which is composed of three parts (on the right side of the equation). The first part expresses the wasted active energy during the task execution. According to the definition, the Critical Speed (CS) is the most power efficient speed with regard to the active power, therefore, the corresponding task is penalized depending on the difference between its current speed and the CS. The second part describes the unbalanced task execution in a cluster. More specifically, the \(t_{\text{unbalanced}}(\tau_i)\) denotes the time (inside a hyper period), where a core executes \(\tau_i\) and all other cores in the same cluster are actually idle but have to operate at the same speed. Obviously, the task with larger \(t_{\text{unbalanced}}(\tau_i)\) should be selected more likely for configuration change. Fig. 1(a) shows one example with two cores in one cluster. In this example we assume that \(S_1\) is assigned to \(\tau_1\) and \(S_2\) to \(\tau_2\), respectively. We observe the time interval between 10ms and 20ms, where the task \(\tau_2\) is executed on the core \(D_{1,2}\) and the core \(D_{1,1}\) also operates at \(S_2\), even though it is idle. In this case the \(t_{\text{unbalanced}}(\tau_2)\) is equal to 10ms. The constants \(\lambda_1\) and \(\lambda_2\) serve as the coefficients to adjust the impact of the first and the second part, respectively. The constant \(\lambda_3\) is a technique parameter in order to prevent the penalty value from being zero. Based on the penalty values we derive the selection probability \(\text{prob}(\tau_i)\) of each task \(\tau_i\) in the Equation 2.

\[
\text{pen}(\tau_i) = \lambda_1 \cdot |F(\text{assign}(\tau_i)) - F(S_{sc})| + \\
\lambda_2 \cdot t_{\text{unbalanced}}(\tau_i) + \lambda_3
\]

(1)

\[
\text{prob}(\tau_i) = \frac{\text{pen}(\tau_i)}{\sum_{\tau_j \in \Gamma} \text{pen}(\tau_j)}
\]

(2)

After a task is selected, in the second step we need to reallocate it by selecting a new core. Hereby we would like to balance the processor load and avoid generating invalid solution, where the system schedulability is not ensured. For this purpose we associate each processor core \(D_{x,y}\) with a reward value \(\text{rew}(D_{x,y})\), which is simply the available utilization that is still free to be used. Formally we define the reward function \(\text{rew}(D_{x,y})\) in the Equation 3.

\[
\text{rew}(D_{x,y}) = \begin{cases} 
U_{ub} - U_{D_{x,y}}, & \text{if alloc}(\tau_k) \neq D_{x,y} \\
U_{ub} - U_{D_{x,y}} + U_k, & \text{otherwise}
\end{cases}
\]

(3)

The \(U_{ub}\) is the upper bound of the total task utilization, so that all tasks are schedulable under a given schedule, e.g. the \(U_{ub}\) is 1 and 0.69 for EDF and RM, respectively. The \(U_{D_{x,y}}\)
is the total task utilization on the core $D_{x,y}$ and the $U_k$ is the utilization of the task $\tau_k$, assuming that the $\tau_k$ is selected in the first step. Analogous to the task selection, based on the reward values we derive the selection probability $\text{prob}(D_{x,y})$ of each core $D_{x,y}$ in the Equation 4.

$$\text{prob}(D_{x,y}) = \frac{\text{rew}(D_{x,y})}{\sum_{D_{i,j}} \text{rew}(D_{i,j})} \quad (4)$$

Intuitively, the core with more available utilization will be selected more likely. Finally, in the third step a new frequency is randomly selected and assigned to the selected task according to the uniform distribution of all possible frequencies. In addition, we take the algorithm $\text{LA+LTF+FF}$ [3] to compute the initial solution of the SA algorithm. The $\text{LA+LTF+FF}$ is a 1.667-approximation algorithm, which attempts to partition the tasks in a similar way as worst fit strategy and assigns all tasks with $S_{xe}$ if possible. Originally it is designed for per-core platforms, however, our preliminary experiments have shown that it is still a good candidate as the starting point in our context.

Algorithm 1 SA based heuristic algorithm

Require: The system model

Ensure: A task partition and a speed assignment

1: Generate initial solution according to $\text{LA+LTF+FF}$
2: Get the value of the initial solution
3: while $\text{iteration} < \text{length}$ do
4: Use the guidance (based on penalty and reward values obtained from the current solution) to generate a neighbour solution as new solution
5: if the new solution is schedulable then
6: Get the value of the new solution
7: Accept new solution and move to the new solution with probability $p$
8: end if
9: $\text{iteration} = \text{iteration} + 1$
10: end while

Algorithm 1 illustrates our main algorithm in detail. In this section we assume that the value of a solution can be obtained in some way and the details are explained later in the subsequent section. In each iteration a neighbour solution is randomly generated based on the penalty and reward values. In case of being schedulable, i.e. the solution can guarantee the system schedulability, which can be checked by means of utilization test, we compare the value of the new solution with the current one. The parameter $\text{length}$ specifies the threshold of the iteration number. Moreover, the new solution is accepted with the probability $p$. In general, the $p$ is dependent on the difference between the values of two solutions. If we denote the value of the current solution and the new solution as $e$ and $e'$, respectively, the acceptance probability is defined in the Equation 5, where $K$ is a constant.

$$p = \begin{cases} 1, & \text{if } e > e' \\ \exp\left(\frac{e-e'}{K}\right), & \text{otherwise} \end{cases} \quad (5)$$

V. Online Execution

In this section we introduce a technique allowing the Algorithm 1 to be executed in an online fashion. This provides several advantages in comparison with the offline fashion: i) it is able to handle dynamic changes in the system, ii) the online evaluation of the solution value is trivial, whereas the offline version requires exact knowledge of the start, preemption and finishing time of each task, which has similar computation complexity as the worst case response time analysis in real-time system, iii) the heuristic information, such as the penalty value of each task, can be updated very easily during runtime, iv) it is able to explore both static and dynamic slack.

In general, the idea is to apply the SA algorithm during runtime. For this we take the advantage of the common feature of the SA algorithm and the real-time system. The SA algorithm is iterative because it iteratively improves the candidate solution; the real-time system with periodic tasks is also iterative in terms of the iteration of task execution in each hyper period. In our approach, one iteration of the SA algorithms is mapped to one hyper period. In other words, in each hyper period we explore and evaluate one candidate solution. This stage is called Exploration Stage (ES) and terminates after a certain number of hyper periods. For the remaining time we name it Application Stage (AS) where the best solution found in the exploration stage is applied. Since the AS is trivial, in the following the explanation focuses on the ES.

The main goal in ES is to find the best solution using the SA algorithm. Our online approach is composed of two parts. The first part happens at each hyper period boundary and the second part occurs at each scheduling point. At each hyper period boundary we mainly perform the work specified for each iteration in the Algorithm 1. More specifically, the current solution is first evaluated (more detail about this will be explained later) and compared with the previous solution. Based on the acceptance probability and the comparison result a decision is made, whether the current solution is accepted or not. Afterwards a neighbour solution is generated for the upcoming hyper period. In order to ensure the system schedulability, if the generated solution is not schedulable, the best known solution so far is used in the upcoming hyper period. In other words, no invalid solution will be selected and used during online execution. Hereby we do not explicitly consider the task migration overhead, because it occurs only at hyper period boundaries. However, if the task migration indeed takes significant time, we can model it as a separate task running always at the maximal frequency and take it into account in the schedulability analysis.

Now we explain the second part performed at each scheduling point. Since our focus is on the cluster-based multi-core platforms, where all cores in the same cluster can only operate at a common frequency, the tasks running in parallel in a
Fig. 1. An example with two cores D1,1 and D1,2 grouped in one cluster. T1 = 20ms, T2 = 40ms. τ1 is partitioned onto D1,1 and τ2 onto D1,2. S1 is assigned to τ1 and S2 to τ2.

The state switching overhead is shown in the Fig. 1 by the width global frequency at each scheduling point. The Fig. 1(a) shows one example, where S2 is assigned to the τ2. In the time interval from 0ms to 10ms the task τ2 is however executed with S1, because S1 has a higher frequency. Subsequently, the task τ1 completes its execution. Thus, in the time interval from 10ms to 20ms the task τ2 is executed at its originally assigned frequency S2. Finally, τ2 completes at 20ms. Clearly, this completion time is earlier than the original completion time of τ2 when it is constantly executed with S2. In this manner, no task will finish later than its WCET and therefore the system schedulability is guaranteed. Moreover, in the time interval from 10ms to 20ms (cf. Fig. 1(a)) the core D1,1 has to execute an idle task at S2, because the core D1,2 is still active and executes a task with S2. Note that this time interval is cumulatively added into the penalty value of the task τ2 during runtime.

If at a scheduling point all cores in the cluster are going to be idle, they can be switched to a low power state (i.e. cluster shutdown). Since the break even time for each low power state is given in the system model, we can easily choose a low power state with the most power saving depending on the length of the upcoming idle interval. Therefore, the main challenge is to decide the arrival time of the next task. In our approach t is retrieved from the head of a sorted queue Q. Basically Q stores the arrival time of each task in sorted manner. Whenever a task is finished, it will update its next arrival time and reinsert it into Q. More precisely, Q is managed by the Algorithm 2 at each scheduling point and its complexity is shown by the lemma 1.

**Lemma 1.** The Algorithm 2 has the computation complexity of $O(\log(n))$ at each scheduling point, where n is the total number of tasks in the system.

**Proof:** According to the line 2 in the Algorithm 2, it is obvious that Q contains the arrival time of each task once at most, which means that the length of Q can never be longer than n. By means of binary search algorithm both the insertion and position finding take $O(\log(n))$. Hence, the computation complexity of Algorithm 2 is $O(\log(n))$ at each scheduling point.

As we consider non-negligible state switching overhead, we need to wake the cores up a little ahead of the actual arrival time of the next task, so that the task will not be delayed due to the state switching latency, which is crucial for meeting the deadlines. This wakeup mechanism is also the reason why the classic schedulability test via utilization can be performed. The state switching overhead is shown in the Fig. 1 by the triangles.

Since we actually run the system under the generated solution, the system energy consumption can be easily recorded during runtime. Additionally, the heuristic information can be easily updated as well. This is one of the major advantages in our online execution. The Algorithm 3 and 4 show the details of energy recording and heuristic information update, respectively.

**Algorithm 2** Manage queue of task arrival times

**Require:** τ, which is finishing at the current scheduling point. Note that there can be at most one task that is finishing at a scheduling point1.

**Ensure:** Updated Q containing time stamps in non-decreasing order.

1: if exists a τ, that is finishing then
2: Get the arrival time $t_{i,next}$ of the next job instance of task $\tau_i$ by updating $t_{i,next} = t_{i,next} + T_i$, and reinsert $t_{i,next}$ into the sorted queue Q.
3: end if
4: Find the position of $t_{current}$ in the Q, and remove all elements before it. The $t_{current}$ denotes the current time.

**Algorithm 3** Energy recording

**Require:** At each scheduling point.

**Ensure:** Updated system energy consumption e.

1: if ls is a P-state then
2: $temp = P(ls) \times (t_{current} - t_{last})$
3: else if ls is a C-state then
4: $temp = P_{on-off}(ls) \times L_{on-off}(ls) + P_{off-on}(ls) \times L_{off-on}(ls) + P(ls) \times (t_{current} - t_{last} - L_{on-off}(ls) - L_{off-on}(ls))$
5: end if
6: $e = e + num \times temp$
7: $t_{last} = t_{current}$
8: Update ls according to the decision made by the activity (marked with A) in the Fig. 2

In the Algorithm 3 and 4, the parameter ls remembers the last processor state during the time interval from the last scheduling point to the current scheduling point. The parameters $t_{last}$ and $t_{current}$ store the time stamps of the last and current scheduling point, respectively. The parameter num gives the number of cores in the current cluster.
the task \( \tau_2 \) is arriving at first time in the current hyper period then
2: \( \text{pen}(\tau) = \lambda_1 \cdot | F(\text{assign}(\tau_1)) - F(S_{sc}) | + \lambda_3 \)
3: end if
4: if there is only one task \( \tau_i \) active in the time interval between the last scheduling point and the current scheduling point then
5: \( \text{pen}(\tau_i) = \text{pen}(\tau_i) + \lambda_2 \cdot (t_{\text{current}} - t_{\text{last}}) \)
6: end if
7: \( t_{\text{last}} = t_{\text{current}} \)

![Algorithm 4 Heuristic information update](image)

**Algorithm 4 Heuristic information update**

**Require:** At each scheduling point.

**Ensure:** Updated heuristic information.

1: if a task \( \tau_i \) is arriving at first time in the current hyper period then
2: \( \text{pen}(\tau) = \lambda_1 \cdot | F(\text{assign}(\tau_i)) - F(S_{sc}) | + \lambda_3 \)
3: end if
4: if there is only one task \( \tau_i \) active in the time interval between the last scheduling point and the current scheduling point then
5: \( \text{pen}(\tau_i) = \text{pen}(\tau_i) + \lambda_2 \cdot (t_{\text{current}} - t_{\text{last}}) \)
6: end if
7: \( t_{\text{last}} = t_{\text{current}} \)

**Theorem 1.** Our online execution has the computation complexity of \( O(\log(n)) \) at each scheduling point, where \( n \) is the number of tasks in the system.

**Proof:** Our online execution contains two main parts. The first part happens at the beginning of each hyper period, where the currently applied solution is evaluated and a new neighbour solution is generated. The evaluation of solution value is solved by runtime recording and the new solution generation needs to change the configuration of only one randomly selected task, which is depending on the heuristic information collected at runtime as well. Afterwards the schedulability test needs to only update the changed task utilization on involved cores (there can be 2 cores getting involved at most). The reward value of involved cores needs to be updated as well. Therefore this part takes \( O(1) \). The second part occurs at each scheduling point and contains all activities shown in the Fig. 2. If we assume that the number of cores in each cluster and the number of available low power states are constant, then the Algorithm 2 clearly dominates other activities. By applying Lemma 1, the complexity of our online execution takes \( O(\log(n)) \) at each scheduling point.

**VI. EVALUATION**

In this section we evaluate the proposed algorithm. Before the results are presented, we first introduce the experiment setup.

**A. Experiment Setup**

Our evaluation is performed by using a SystemC RTOS simulation framework [14]. We extended their library by adding the additional support for multi-core simulation with DPM/DVS capabilities. Since we are interested in cluster-based multi-core processor platforms, different configurations are investigated, e.g. the platform with 4 cores grouped into 2 clusters or into 1 cluster. In this experiment setup we concentrate on symmetry cluster-based platforms, where all clusters contain the same number of cores. However, our algorithm is not constrained to symmetry configuration. Furthermore, for the sake of simplicity we assume that all cores possess the same power model. In the following test scenarios we
refer to the platform with \( X \) cores grouped into \( Y \) clusters as core\(X\)cluster\(Y\). For each core we adopt the power model of Intel PXA270 processor [1], which supports 5 P-states (from 208MHz to 624MHz) and 3 C-states (active, idle and sleep).

Based on this experiment setup we performed three test scenarios. In each test scenario we applied EDF as real-time task schedule on each core. The first two scenarios deal with synthetic task sets and the third scenario considers real life case studies. According to our preliminary experiments, the parameters of our algorithm are set as follows: \( \lambda_1 = 10 \), \( \lambda_2 = 1^3 \), \( \lambda_3 = 1 \) and \( p = \exp\left(\frac{-c - p}{100}\right) \). Finally, in the last subsection the actual scheduling overhead is discussed.

B. Investigating Static Slack

In this subsection we evaluate the behaviour of our algorithm with the assumption that all tasks run until the WCET. For this we randomly generated 1000 task sets and the size of each task set is between 5 and 20. The period of each task is within \([0.05ms, 20ms]\) and the utilization of each task set is within \([0.4, 4.5]\). In this test scenario we executed the task sets on the processor platforms with 4 cores, 8 cores and 16 cores, respectively. The Fig. 3 illustrates the simulation results, where the y-axis shows the system energy consumption normalized with regard to the initial solution \((LA+LTF+FF)\). The legend \(SA\) represents the solution obtained by the generalized \(SA\) and the \(SA+H\) represents our algorithm, which considers heuristic information. Additionally, the threshold of the iteration number (the parameter \( length \) in the Algorithm 1) is set to 1000, 2000 and 4000 for the 4-core, 8-core and 16-core platforms, respectively. This experiment shows that our approach achieves up to 15% power reduction in comparison with the algorithm \(LA+LTF+FF\). Furthermore, the results also confirm our expectation, where the \(SA+H\) outperforms other algorithms on all platforms. Actually, we also should compare our algorithm with the algorithms presented by Kong et al. [11] and Qi et al. [12], because they are the closest related work. However, their algorithms are designed for the frame-based task model and therefore not applicable in our case.

C. Investigating Dynamic Slack

In this test scenario our focus is on the impact of task runtime variation, which is the actual execution time of the tasks. We used the same task sets from previous subsection. However, during runtime we simulate the runtime variation by altering the execution time of each task according to the 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 \(\alpha\) is defined by \(B_i = W_i \cdot (1 - \alpha)\) with \(0 < \alpha < 1\). The Gaussian function is then defined with dependence to \(W_i\) and \(\alpha\), namely the mean is \(\mu = \frac{W_i + B_i}{2} = W_i \cdot \frac{1 - \alpha}{2}\) and the variance is \(\sigma^2 = 0.4 \cdot \frac{W_i - B_i}{2} = 0.2 \cdot \alpha \cdot W_i\). In other words, the greater the \(\alpha\) value, the more dynamic slack can be utilized and the more power we can save. Moreover, the per-core platforms achieve the most power reduction, because they provide the most flexibility. The experiment on the 8-core platforms shows similar result, which is shown in the Fig. 5.

D. Real Life Case Studies

In order to make the evaluation more realistic, we simulated 4 real life case studies taken from [15]. The first case study is a linear motor control (LMC) system, which contains 2 periodic tasks for current control and 1 periodic task for speed control. The second test case is an industrial automation example, a CNC machine, which is composed of three subsystems being responsible for the human machine interaction, database access and sensor/actuator control, respectively. The third test scenario is an X-ray control system, which controls the
TABLE I
SIMULATION RESULTS OF REAL CASE STUDIES

<table>
<thead>
<tr>
<th>Case study</th>
<th>Task number</th>
<th>Hyper period (ms)</th>
<th>Power saving</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMC</td>
<td>3</td>
<td>42</td>
<td>48%</td>
</tr>
<tr>
<td>CNC</td>
<td>12</td>
<td>100</td>
<td>3.5%</td>
</tr>
<tr>
<td>X-ray</td>
<td>8</td>
<td>1000</td>
<td>17.2%</td>
</tr>
<tr>
<td>ACDA</td>
<td>19</td>
<td>300</td>
<td>12.3%</td>
</tr>
</tbody>
</table>

actuators to position the X-ray beam properly. The last case study is a control system in the automotive area (ACAD), which consists of an airbag control unit, an anti-lock braking system and an electronic stability control system. More detail about the task set of the case studies can be found in the Table I. We simulated all of them on a platform with 4 cores divided into 2 clusters. TABLE I shows the simulation results under the assumption that all tasks run until the WCET. The power reduction is acquired by comparing the solution found after 1000 hyper periods using our algorithm with the initial solution. For the test case LMC it shows that the solution found by our algorithm is in fact optimal in terms of the power consumption. In general, our approach may suffer, if the hyper period is arbitrarily long. However, these case studies [15] show that the task periods are usually harmonic in the real life, which means that the length of the hyper periods is usually manageable.

E. Scheduling Overhead

Even though the theoretical complexity of our approach has been proven in the Theorem 1, we are also interested in the actual scheduling overhead. Since the evaluation is performed through simulation, we compared the overhead introduced by our approach with the overhead caused by the traditional EDF scheduling algorithm implemented in the SystemC RTOS simulation framework [14]. More specifically, we denote $\rho_1$ and $\rho_2$ by the Equation 6 and 7, respectively. $\rho_1$ reflects the ratio between the overhead coming from the Algorithm 1 and the EDF overhead. Since at each hyper period boundary we execute one iteration of the Algorithm 1, hereby we measured how long one iteration will take in average in our implementation. The $\rho_2$ gives the average ratio between the total overhead of the activities shown in the Fig. 2 and the EDF overhead at each scheduling point. The experiment results show $\rho_1 = 0.72$ and $\rho_2 = 1.81$, which means that the actual scheduling overhead introduced by our approach is in the same order of magnitude as the EDF overhead.

$$\rho_1 = \frac{\text{our overhead at each hyper period boundary}}{\text{EDF overhead at each scheduling point}}$$  \hspace{1cm} (6)

$$\rho_2 = \frac{\text{our overhead at each scheduling point}}{\text{EDF overhead at each scheduling point}}$$  \hspace{1cm} (7)

VII. CONCLUSION

With the continuous increasing of system complexity and the technology advance towards multi-core processor platforms, the problem of energy efficiency becomes more and more difficult. In this article we focused on scheduling hard real-time tasks on cluster-based multi-core processor platforms. The well-established techniques DPM and DVS are applied together to minimize the system power consumption. We proposed a simulated annealing based heuristic algorithm and its online execution with logarithmic complexity at each scheduling point. Our approach is able to deal with multiple DPM low power states with non-negligible state switching overhead. Furthermore, both static slack and dynamic slack are explored and utilized. Finally, through the experiment results our approach shows its great energy efficiency in comparison with the existing algorithms.

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