Energy-Efficient Scheduling on Homogeneous Multiprocessor Platforms

Jian-Jia Chen  
Computer Engineering and Networks Lab. (TIK)  
ETH Zurich, Switzerland  
jchen@tik.ee.ethz.ch

Lothar Thiele  
Computer Engineering and Networks Lab. (TIK)  
ETH Zurich, Switzerland  
thiele@tik.ee.ethz.ch

ABSTRACT

Low-power and energy-efficient system implementations have become very important design issues to extend operation duration or cut power bills. To balance the energy consumption resulting from the dynamic power consumption and the static power consumption, the concept of critical speed has been adopted widely in the literature. Most scheduling algorithms for such systems assume that the critical speed is the lowest speed for scheduling and then perform job/task procrastination to turn the processor(s) to the dormant mode when there is no job for execution. This paper shows that the critical speed might be too optimistic and turning the processor(s) to the dormant mode might be energy-inefficient. By allowing tasks to run at lower speeds than the critical speed, in this paper, a new approximation algorithm is developed for homogeneous multiprocessor systems with a 1.21-approximation factor, which significantly improves the state-of-the-art approximation algorithm with a 1.667-approximation factor. Performance evaluation shows the effectiveness of the proposed algorithm with comparison to the state-of-the-art approximation algorithm. Our algorithm can reduce the energy consumption by at most 15% in our simulations.

Categories and Subject Descriptors

General Terms
Algorithms, Performance

Keywords
static power consumption, dynamic voltage scaling, task partitioning, real-time scheduling.

1. INTRODUCTION

Power management has become an important system design issue for embedded systems and server systems to prolong the operation duration or reduce power bills. Dynamic power consumption (mainly) due to switching activities and static power consumption (mainly) due to leakage current are two major sources of energy consumption of a CMOS processor [8]. The dynamic voltage scaling (DVS) technique was introduced to balance the dynamic energy consumption and the performance of a system, in which different supply voltages lead to different execution speeds/frequencies. The dynamic power consumption is usually a convex and increasing function of speed/frequency, which motivates to execute as slowly as possible. However, executing at a lower speed stretches the execution time with more leakage/static energy consumption.

For systems with real-time demands, energy-efficient scheduling in DVS systems has been studied extensively. For real-time systems, energy-efficient scheduling is to minimize the energy consumption under timing constraints. As chip makers are releasing multi-core chips and multiprocessor system-on-a-chip (MPSoC), multiprocessor platforms have become even more popular to improve the system performance and accommodate the growing demand of application functionality.

In nano-meter manufacturing, static power consumption in CMOS circuits has significant contribution of the total power consumption, in which the static power consumption is comparable to the dynamic power dissipation [8]. To reduce the static energy consumption resulting from the leakage current, a processor might enter a dormant mode (or be turned off), in which the power consumption of the processor in the dormant mode is much smaller than the static power consumption. However, turning the processor to the active mode requires time and energy overheads, due to the wakeup/shutdown of the processor and data fetch in the register/cache. For example, the Transmeta processor in 70nm technology has 483μJ energy overhead and less than 2 msec (ms) timing overhead [8].

Energy-efficient scheduling has been recently explored on DVS platforms with non-negligible static power consumption, such as [2, 4, 8, 9], in which the static power can be reduced by turning the processor to the dormant mode. In particular, for scheduling periodic tasks in uniprocessor systems, Chen and Kuo [2], Jejurikar et al. [7, 8], and Lee et al. [9] develop DVS and procrastination scheduling strategies to decide the execution speeds and when to turn the processor to the dormant mode. For uniprocessor scheduling of aperiodic real-time tasks, Irani et al. [5] propose a 3-approximation algorithm for the minimization of energy consumption, where a ρ-approximation algorithm guarantees to derive solutions no more than ρ times of the optimal energy consumption. Niu and Quan [11] apply similar procrastination strategies for periodic real-time tasks by considering the hyper-period of the given tasks. For homogeneous multiprocessor systems, Xu et al. [13] propose algorithms to determine the number of activated processors by considering workload requests instead of real-time tasks, and de Langen and Juurlink [4] provide heuristic algorithms for systems with discrete speeds. Chen et al. [1] develop a polynomial-time approximation algorithm for energy consumption minimization.

This work is sponsored in part by the European Community’s Seventh Framework Programme FP7/2007-2013 project Predator (Grant 216008).
Most of the above approaches [1,2,5,7,8,11,13] apply the critical speed as the lower-bounded speed for execution and then perform job/task procrastination without violating the timing constraints, in which the critical speed is the available speed with the minimum energy consumption for execution. By definition, if $P(s)$ is the power consumption at speed $s$, the critical speed $s^*$ is the speed $s$ with the minimum energy consumption $P(s)/s$. However, as the critical speed is defined only for the minimization of energy consumption for job execution, it, in fact, optimistically assumes to turn the processor(s) to the dormant mode after the execution. However, it might not be worthy to turn the processor to the dormant mode because the energy overhead might be larger than the reduced energy. In such cases, executing at the critical speed might lead to a solution that consumes more energy. Furthermore, the critical speed is used in [1] as the maximum speed for task assignment and processor allocation. As executing at the critical speed might consume more energy, task assignment based on the critical speed with load balancing might also consume more energy.

To clarify the non-optimality of critical speeds, we will first explore energy-efficient scheduling for frame-based real-time tasks, in which every task has the same deadline and period. For the simplicity of presentation, we assume $P(s)/s$ is merely a convex function [5]. For example, the most adopted speed-dependent power consumption function $P_0(s) = \alpha s^3 + \lambda s$ satisfies the assumption for any $\gamma > 1$ and any positive constants $\alpha$ and $\lambda$. The critical speed $s^*$, defined as the available speed that minimizes the energy consumption for execution [2,5,8], is the speed $s$ with the minimum $P_0(s)/s$.

For example, when $P_0(s) = \alpha s^3 + \lambda s$, the critical speed $s^*$ is $\max\left\{\min\left\{\sqrt[3]{\frac{\alpha}{\lambda}}, s_{\max}\right\}, \min\right\}$. A processor has two modes: dormant mode and active mode. When the processor is in the dormant mode, the power consumption of the processor is assumed to be 0 by scaling the static power consumption [2,5], but the results in this paper still hold when the power consumption in the dormant mode is not 0. The processor has to be in the active mode for task execution. However, switching between the two modes takes time and consumes energy. Since periodic tasks are considered, the procedure to turn the processor to the dormant mode can be assumed instantaneously with negligible energy overhead by treating the overhead as a part of the overhead to turn on the processor. We denote $E_{sw}(t_{sw})$, respectively as the energy (time, respectively) of the switching overhead from the dormant mode to the active mode.

When the processor is idle in the active mode, the processor executes NOP instructions at speed $s_{\min}$ for energy minimization.

2. SYSTEM MODELS

2.1 Processor models

The (system) power consumption function $P(s)$ of speed $s$ on a processor has two parts: $P_0(s)$ and $P_{\text{nod}}$, where $P_0(s)$ ($P_{\text{nod}}$, respectively) is dependent (independent, respectively) on the speed [2,14]. The speed-dependent power consumption $P_0(s)$ comes from the short-circuit power consumption and the dynamic power consumption due to the charging and discharging of gates on a CMOS DVS processor, while the leakage power consumption mainly contributes to $P_{\text{nod}}$. For example, the dynamic power consumption $P_{\text{switch}}(s)$ due to gate switching at speed $s$ is $P_{\text{switch}}(s) = C_{ef}V_{dd}s$, where $s = k(V_{dd} - V_{th})^2$, and $C_{ef}$, $V_{dd}$, $V_{th}$, and $k$ denote the effective switch capacitance, the threshold voltage, the supply voltage, and a design-specific constant, respectively ($V_{dd} \geq V_{th} \geq 0, k > 0$, and $C_{ef} > 0$). The speed-dependent power consumption function $P_0(s)$ can be modeled as a strictly convex and increasing function of the adopted speed, while the speed-independent power consumption is a constant. We explore the scheduling on processors with a continuous spectrum of the available speeds between the upper-bounded speed $s_{\max}$ and the lower-bounded speed $s_{\min}$. It is easy to apply the algorithms in [6] to derive the devised solutions to systems with discrete speeds only.

We assume that $P(s)$ is a strictly convex and increasing function, while $P(s)/s$ is merely a convex function [5]. For example, the most adopted speed-dependent power consumption function $P_0(s) = \alpha s^3 + \lambda s$ satisfies the assumption for any $\gamma > 1$ and any positive constants $\alpha$ and $\lambda$. The critical speed $s^*$, defined as the available speed that minimizes the energy consumption for execution [2,5,8], is the speed $s$ with the minimum $P_0(s)/s$.

For example, when $P_0(s) = \alpha s^3 + \lambda s$, the critical speed $s^*$ is $\max\left\{\min\left\{\sqrt[3]{\frac{\alpha}{\lambda}}, s_{\max}\right\}, \min\right\}$. A processor has two modes: dormant mode and active mode. When the processor is in the dormant mode, the power consumption of the processor is assumed to be 0 by scaling the static power consumption [2,5], but the results in this paper still hold when the power consumption in the dormant mode is not 0. The processor has to be in the active mode for task execution. However, switching between the two modes takes time and consumes energy. Since periodic tasks are considered, the procedure to turn the processor to the dormant mode can be assumed instantaneously with negligible energy overhead by treating the overhead as a part of the overhead to turn on the processor. We denote $E_{sw}(t_{sw})$, respectively as the energy (time, respectively) of the switching overhead from the dormant mode to the active mode.

When the processor is idle in the active mode, the processor executes NOP instructions at speed $s_{\min}$ for energy minimization.

When the processor is idle and the idle interval is longer than the break-even time $t_{sw}(s_{\min})$, turning it to the dormant mode is worthwhile. Let $t_0$ be the break-even time, i.e., $t_0 = \frac{E_{sw}(t_{sw})}{P_0(s_\min)}$. Without loss of generality, we assume that $t_{sw}$ is no more than $t_0$.

We explore energy-efficient scheduling on $m$ homogeneous DVS multiprocessors, where the power consumption function of each task is the same for every processor. For brevity, we denote these $m$ processors by $M_1, M_2, \ldots, M_m$.

For the simplicity of presentation, we assume $P_0(s) = \alpha s^3 + \lambda s$ for constants $\alpha$ and $\lambda$, where $P_{\text{nod}}$ is abbreviated by $\beta$. The proposed algorithm can be extended to any other convex and increasing functions of $P_0(s)$ easily. Moreover, for simplicity, we implicitly take $s_{\min} = 0$ and $s_{\max} = \infty$ if there is no specific statement. How to handle the case that $s_{\min} \neq 0$ and $s_{\max} \neq \infty$ will also be presented at the end of Section 4. Moreover, we do not intend to deal with the schedulability of real-time tasks under the speed constraint. As shown in [1], it is $\mathcal{NP}$-complete to determine whether there exists a feasible solution when $s_{\max} \neq \infty$. Generally, the maximum available speed is quite higher than the critical speed.

2.2 Task models

We explore the scheduling of a set of periodic tasks in a multi-processor DVS system. A periodic task is an infinite sequence of task instances, referred to as jobs, where each job of a task comes in a regular period. The period of task $T_j$ is denoted by $p_j$. The relative deadline $d_j$ of task $T_j$ is equal to its period $p_j$. The amount of the worst-case execution cycles of task $T_j$ is profiled and denoted by $c_j$. The value of $c_j$ could also be obtained by profiling the
worse-case execution time at a certain profiling speed.

Executing at speed \( s \) for time units is assumed to complete at speed \( s \times t \) with energy consumption \( P(s)t \). Without loss of generality, we are only concerned with the case that \( p_j > 0 \), in which the tasks can complete before their deadline at speed \( \lambda_{\text{max}} \).

Note that, for a task \( \tau_i \), no matter which speeds the scheduler uses for executing a task instance of \( \tau_i \), the (partial) energy consumption comes from the power consumption term \( \lambda s \) which does not relate to the schedule. As a result, for algorithmic design and analysis, we will simply assume that \( \lambda = 0 \), which does not affect the applicability of the designed approach and analysis for the general case \( \lambda > 0 \).

When all the tasks are with the same period \( D \) and arrival time \( 0 \), these tasks are called frame-based real-time tasks with frame size \( D \). As shown in the literature [10], we can transform frame-based real-time tasks with precedence constraints to independent frame-based real-time tasks by pipelining the execution of the tasks. The proposed algorithm in this paper can then be adopted. Therefore, throughout this paper, we only focus on tasks without precedence constraints.

### 2.3 Problem definition

The problem explored in this paper is defined as follows:

Consider a set \( T \) of independent tasks over \( m \) identical processors with a common power consumption function \( P(s) = \alpha s^\alpha + \beta \), where \( \alpha, \beta \geq 0 \). Each of the processors can operate at any speed in the range of \( \lambda_{\text{min}} \) and \( \lambda_{\text{max}} \). Each periodic task \( \tau_j \in T \) is associated with a computation requirement in \( c_j \) CPU-cycles and a period \( p_j \), where the relative deadline of \( \tau_j \) is \( p_j \). The energy of the switching overheads from the dormant mode to the active mode of a processor is \( E_{\text{sw}} \), while the timing overhead is \( t_{\text{sw}} \leq \frac{E_{\text{sw}}}{\lambda_{\text{min}}} \).

The objective is to partition tasks in \( T \) to \( m \) processors so that the energy consumption is minimized and tasks can be done before their timing constraints.

For brevity, the studied problem is referred to as the leakage-aware multiprocessor energy-efficient scheduling (LAMS) problem, which has been shown to be \( \mathcal{NP} \)-hard in a strong sense in [1]. This paper pursues polynomial-time approximation algorithms with worst-case guarantees on the quality of the derived solutions for the LAMS problem. A \( \rho \)-approximation algorithm for the LAMS problem (or, an algorithm with a \( \rho \)-approximation factor) derives solutions with at most \( \rho \) times of the corresponding optimal solutions.

### 3. CRITICALITY OF CRITICAL SPEEDS

This section presents the reasons why the critical speed should not be used as the only lower bound for execution or the only upper bound for task assignment. We then present the weakness of the approximation algorithm in [1] by providing a tight example.

#### 3.1 Non-optimality of critical speeds

Throughout this subsection, we will demonstrate the weakness of the critical speed by using the power consumption function for Intel XScale, in which the power consumption can be approximately modeled as \( P(s) = (1.52 \cdot \text{MHz})^2 + 0.08 \) Watt [2, 12]. For such a case, the critical speed \( s^* \) is around 297MHz and the power consumption at the critical speed is 0.12Watt where \( \lambda_{\text{min}} \) is assumed 0 in this example. We assume that the switching overhead \( E_{\text{sw}} \) is 0.8 mJoule. Therefore, the break-even time \( t_\theta \) is 10 msec, provided that the idle power consumption of the processor is 0.08 Watt. We will only use frame-based real-time tasks with a common

\[ \begin{align*}
0 & \leq p_j \leq \frac{1}{\alpha} \\
0 & \leq \lambda \leq \lambda_{\text{max}} \\
0 & \leq \lambda \leq \lambda_{\text{max}}
\end{align*} \]

Figure 1: Different schedules on one processor.

![Figure 1](image)

Figure 2 depicts the energy consumption in time interval \( (0, 30) \) msec when the amount of computation of the task is between 0 and 0.03s

Figure 2(a) shows the energy consumption for two different scheduling strategies: (1) the fully-utilized strategy is to execute at the speed to fully utilize the time interval \( (0, 30) \) to minimize the energy consumption resulting from the dynamic power circuits, and (2) the critical-speed strategy is to execute at the critical speed and then either turn the processor to the dormant mode or be idle depending on the completion time. Figure 2(b) is the minimum energy consumption between the above strategies. As shown in Figure 2, in this example, executing at the critical speed is better only when the amount of the computation requirement is no more than 0.0125s

It seems to have benefit when the amount of the computation requirement is more than 0.03s by executing at the critical speed on two processors. But, it is not always true, again. Consider the input instance with two tasks \( \tau_1 \) and \( \tau_2 \) when \( m = 2 \). Figure 3 shows the schedules for the above tasks when \( c_1 = 0.03s \) and \( c_2 = 0.005s \) by using two processors (in Figure 3(a)) and by using one processor only (in Figure 3(b)). The energy consumption for the schedule in Figure 3(a) (Figure 3(b), respectively) is 5.431, respectively) mJoule. To see when to use two processors, suppose that the total amount \( C = c_1 + c_2 \) of computation cycles required by these two tasks is between 0 and 0.05s

1 The energy consumption is defined by executing these tasks in a certain interval \( L \).

If the hyper-period of the tasks exists, \( L \) should be defined as the hyper-period. Otherwise, \( L \) should be a number that is large enough to show the representative.
Suppose that after breaking ties arbitrarily. Clearly, Algorithm LALTF does not perform well when the system is not highly loaded at the critical speed such as the example in Section 3.2.

For notational brevity, let tasks be indexed from the largest to the smallest by breaking ties arbitrarily. Clearly, Algorithm LALTF will assign the first m tasks to m different processors. Suppose that after assigning task τ_k, the following conditions are satisfied: \( \frac{c_k}{p_k} \geq s^* \), \( \frac{c_k}{p_k} \leq s^* \), and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < (m - k) \). It is not difficult to prove that there exists a lower-bounded solution which executes task \( \tau_j \) for \( j = 1, 2, \ldots, k \) at speed \( \frac{c_j}{p_j} \). As a result, what we have to do is to decide how to schedule tasks \( \{ \tau_{k+1}, \tau_{k+2}, \ldots, \tau_{\ell_k} \} \) on \( m - k \) processors. If there does exist such a task \( \tau_k \), applying Algorithm LALTF can yield to a 1.13-approximation solution.

As a result, we only discuss the other case that such a task \( \tau_k \) exists or \( k = 0 \) (i.e., \( \frac{c_k}{p_k} \leq s^* \) and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < m \)). To simplify the presentation, we will abuse and override the symbol \( m \) and information is negligible, applying Algorithm LALTF in such a case, is marginal, and can be ignored. When speed-independent power consumption is negligible, applying Algorithm LALTF has a 1.13-approximation factor \[1\]. Algorithm LALTF does not perform well when the system is not highly loaded at the critical speed such as the example in Section 3.2.

For notational brevity, let tasks be indexed from the largest to the smallest by breaking ties arbitrarily. Clearly, Algorithm LALTF will assign the first m tasks to m different processors. Suppose that after assigning task \( \tau_k \), the following conditions are satisfied: \( \frac{c_k}{p_k} \geq s^* \), \( \frac{c_k}{p_k} \leq s^* \), and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < (m - k) \). It is not difficult to prove that there exists a lower-bounded solution which executes task \( \tau_j \) for \( j = 1, 2, \ldots, k \) at speed \( \frac{c_j}{p_j} \). As a result, what we have to do is to decide how to schedule tasks \( \{ \tau_{k+1}, \tau_{k+2}, \ldots, \tau_{\ell_k} \} \) on \( m - k \) processors. If there does exist such a task \( \tau_k \), applying Algorithm LALTF can yield to a 1.13-approximation solution.

As a result, we only discuss the other case that such a task \( \tau_k \) exists or \( k = 0 \) (i.e., \( \frac{c_k}{p_k} \leq s^* \) and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < m \)). To simplify the presentation, we will abuse and override the symbol \( m \) and information is negligible, applying Algorithm LALTF in such a case, is marginal, and can be ignored. When speed-independent power consumption is negligible, applying Algorithm LALTF has a 1.13-approximation factor \[1\]. Algorithm LALTF does not perform well when the system is not highly loaded at the critical speed such as the example in Section 3.2.

For notational brevity, let tasks be indexed from the largest to the smallest by breaking ties arbitrarily. Clearly, Algorithm LALTF will assign the first m tasks to m different processors. Suppose that after assigning task \( \tau_k \), the following conditions are satisfied: \( \frac{c_k}{p_k} \geq s^* \), \( \frac{c_k}{p_k} \leq s^* \), and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < (m - k) \). It is not difficult to prove that there exists a lower-bounded solution which executes task \( \tau_j \) for \( j = 1, 2, \ldots, k \) at speed \( \frac{c_j}{p_j} \). As a result, what we have to do is to decide how to schedule tasks \( \{ \tau_{k+1}, \tau_{k+2}, \ldots, \tau_{\ell_k} \} \) on \( m - k \) processors. If there does exist such a task \( \tau_k \), applying Algorithm LALTF can yield to a 1.13-approximation solution.

As a result, we only discuss the other case that such a task \( \tau_k \) exists or \( k = 0 \) (i.e., \( \frac{c_k}{p_k} \leq s^* \) and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < m \)). To simplify the presentation, we will abuse and override the symbol \( m \) and information is negligible, applying Algorithm LALTF in such a case, is marginal, and can be ignored. When speed-independent power consumption is negligible, applying Algorithm LALTF has a 1.13-approximation factor \[1\]. Algorithm LALTF does not perform well when the system is not highly loaded at the critical speed such as the example in Section 3.2.

For notational brevity, let tasks be indexed from the largest to the smallest by breaking ties arbitrarily. Clearly, Algorithm LALTF will assign the first m tasks to m different processors. Suppose that after assigning task \( \tau_k \), the following conditions are satisfied: \( \frac{c_k}{p_k} \geq s^* \), \( \frac{c_k}{p_k} \leq s^* \), and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < (m - k) \). It is not difficult to prove that there exists a lower-bounded solution which executes task \( \tau_j \) for \( j = 1, 2, \ldots, k \) at speed \( \frac{c_j}{p_j} \). As a result, what we have to do is to decide how to schedule tasks \( \{ \tau_{k+1}, \tau_{k+2}, \ldots, \tau_{\ell_k} \} \) on \( m - k \) processors. If there does exist such a task \( \tau_k \), applying Algorithm LALTF can yield to a 1.13-approximation solution.

As a result, we only discuss the other case that such a task \( \tau_k \) exists or \( k = 0 \) (i.e., \( \frac{c_k}{p_k} \leq s^* \) and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < m \)). To simplify the presentation, we will abuse and override the symbol \( m \) and information is negligible, applying Algorithm LALTF in such a case, is marginal, and can be ignored. When speed-independent power consumption is negligible, applying Algorithm LALTF has a 1.13-approximation factor \[1\]. Algorithm LALTF does not perform well when the system is not highly loaded at the critical speed such as the example in Section 3.2.

For notational brevity, let tasks be indexed from the largest to the smallest by breaking ties arbitrarily. Clearly, Algorithm LALTF will assign the first m tasks to m different processors. Suppose that after assigning task \( \tau_k \), the following conditions are satisfied: \( \frac{c_k}{p_k} \geq s^* \), \( \frac{c_k}{p_k} \leq s^* \), and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < (m - k) \). It is not difficult to prove that there exists a lower-bounded solution which executes task \( \tau_j \) for \( j = 1, 2, \ldots, k \) at speed \( \frac{c_j}{p_j} \). As a result, what we have to do is to decide how to schedule tasks \( \{ \tau_{k+1}, \tau_{k+2}, \ldots, \tau_{\ell_k} \} \) on \( m - k \) processors. If there does exist such a task \( \tau_k \), applying Algorithm LALTF can yield to a 1.13-approximation solution.

As a result, we only discuss the other case that such a task \( \tau_k \) exists or \( k = 0 \) (i.e., \( \frac{c_k}{p_k} \leq s^* \) and \( \sum_{j=k+1}^{\ell_k} \frac{c_j}{p_j} < m \)). To simplify the presentation, we will abuse and override the symbol \( m \) and information is negligible, applying Algorithm LALTF in such a case, is marginal, and can be ignored. When speed-independent power consumption is negligible, applying Algorithm LALTF has a 1.13-approximation factor \[1\]. Algorithm LALTF does not perform well when the system is not highly loaded at the critical speed such as the example in Section 3.2.
4.1 Algorithm for frame-based tasks

For notational brevity, let \( D \) be the common deadline of the frame-based real-time tasks. Our approach is to apply the largest-task-first algorithm again, but not on \( m \) processors. The idea is to use either \( \sum_{T \in T} \frac{c_j}{T} + 1 \) processors or only \( \sum_{T \in T} \frac{c_j}{T} \) processors. As \( \sum_{T \in T} \frac{c_j}{T} < m \), the above assignment does not violate the number of available processors.

For brevity, let \( m^* \) be \( \sum_{T \in T} \frac{c_j}{T} \). As a result, we will only focus on two cases: one is to use \( m^* \) processors for scheduling tasks in \( T \), and the other is to use \( m^* + 1 \) processors. For these two cases, we greedily apply the largest-task-first strategy to assign tasks onto those processors. Then, each processor individually minimizes its energy consumption by (1) executing at the critical speed and then turning to the dormant mode\(^2\), or (2) executing at a constant speed without idling or turning to the dormant mode. Between these two schedules, the better one (with the less energy consumption) is picked as the solution.

Algorithm 1 illustrates the pseudo-code of the above algorithm, denoted by Algorithm RSLTF. The time complexity of Algorithm RSLTF is \( O(|T| \log (m|T|)) \), since it takes \( O(|T| \log |T|) \) for the sorting of task set \( T \) and \( O(|T| \log m) \) for assigning tasks in the loop between Step 4 and Step 7 in Algorithm 1.

Note that Algorithm RSLTF is different from Algorithm LALTF developed in [1]. The difference is as follows: Algorithm RSLTF tries to assign tasks on either \( m^* \) or \( m^* + 1 \) processors by applying the largest-task-first strategy without setting any workload constraints on these processors. Algorithm RSLTF uses the largest-task-first strategy to assign tasks on \( m \) processors and then performs task re-assignment by setting \( s^* \) as the workload constraints on these processors. Take Figure 3 in Section 3 for two tasks for example. Algorithm RSLTF allocates two processors for task execution if \( 3z^2 > 1.285z^2 \); otherwise, only one processor is used for executing the tasks.

We now present the performance analysis of Algorithm RSLTF in terms of approximation factors. If \( m^* = 0 \), the LAMS problem is equivalent to a uniprocessor scheduling problem, and, hence, the derived solution is optimal for frame-based real-time tasks.

For the case that \( m^* > 0 \), multiprocessor platforms might be used in optimal solutions of the LAMS problem. We will analyze the approximation factor of Algorithm RSLTF for two cases: \( m^* = 1 \) and \( 2 \leq m^* \). The following lemma of the property of the largest-task-first strategy will be widely used for the analysis of approximation factors.

**Lemma 1.** Applying the largest-task-first strategy for task partitioning of \( T \) on \( m^* \) processors with \( m^* \leq |T| \) and \( m^* < s^* \) for all tasks \( T \) in \( T \), the largest workload of these \( m^* \) processors is at most twice of the smallest workload of these \( m^* \) processors if the largest workload is no less than \( s^* \), where the workload on a processor is defined as the summation of the computation requirements of the assigned tasks divided by the common deadline \( D \).

**Proof.** As the processor with the largest workload must be no less than \( s^* \), the difference between the maximum workload and the minimum workload is at most the workload of the task assigned to the processor with the maximum workload. As a result, this lemma is proved.

4.1.1 Analysis when \( m^* = 1 \)

When \( m^* = 1 \), Algorithm RSLTF allocates either one processor or two processors. For brevity, let \( z = \sum_{T \in T} \frac{c_j}{T} \). The energy consumption for executing all the tasks at the critical speed is the lower bound of the optimal solution for this case. As a result, the lower bound of the optimal solution is \( 1.5\beta zD \), where \( 1.5\beta \) is the power consumption at the critical speed.

For the case that only one processor is allocated, the energy consumption is \( D(\alpha(zs^*)^3 + \beta) = D(0.5z^3 + \beta) \), since \( s^* = \frac{1}{z} \). We now focus on the case that the derived solution of Algorithm RSLTF uses two processors. Suppose that \( z_1, z_2 \) (\( z_2^* \), respectively) is the workload on the first (second, respectively) allocated processor for execution. Without loss of generality, we assume that \( z_1 \leq z_2 \). Here are two cases. If \( z_2 > 1 \), we know that \( 1 \leq z_2 \leq 2z_1 \) based on Lemma 1. Executing at speed \( z_1^* \) on the first processor completes all the tasks on this processor in time with energy consumption \( D(0.5z_1^3 + 1) \). Similarly, the energy consumption on the second processor in this case is \( D(0.5(z_2^3 + 1)) \). If \( z_2 \leq 1 \), the energy consumption is at most \( 1.5D(\beta z_2 + (2 - z)D) \) by executing at the critical speed without turning the processors to the dormant mode.

As there are at most two processors allocated for task execution, we also know that \( z_1 + z_2 = z \). Therefore, we know that the approximation factor \( \rho \) for this case is

\[
\rho \leq \max_{z \geq 2} \min \left\{ \frac{3^2 + 2}{3z}, \max \left\{ \frac{4 + z}{3} \frac{(z^3 + z^2)}{3z} \right\} \right\} \tag{1}
\]

Clearly, function \( \frac{3^2 + 2}{3z} \) is a monotonically increasing function of variable \( z \) when \( z \geq 1 \), while \( \frac{z^3}{3z^2} \) is a monotonically decreasing function. However, the exact value of \( \max_{z \geq 2} \min \) depends on the value of \( z \), \( z_1 \), and \( z_2 \), and is difficult to evaluate. Fortunately, it is possible to have an upper bound by applying the convexity of function \( z^3 \) and the relationship of \( z_1 \) and \( z_2 \) based on Lemma 1. Because of the convexity, it is known that \( x_1^3 + x_2^3 \geq z_1^3 + z_2^3 \) if \( x_1 \leq z_1 \leq x_2 \leq z_2 \) and \( z_1 + z_2 = x_1 + x_2 \) for any \( x_1, x_2 \geq 0 \). As \( z_2 \leq 2z_1 \), we know that \( x_1^3 + (2x_1)^3 \geq z_1^3 + z_2^3 \) for \( x_1 = 2z_1 \). As a result, we have

\[
\frac{(z_1^3 + z_2^3) + 4}{3z} \leq \frac{(z_1^3 + z_2^3) + 4}{3z} \leq \frac{1}{z} \frac{z^3 + 4}{3z} = \frac{1}{z} \frac{z^3 + 4}{3z} \tag{2}
\]

**Lemma 2.** When \( m^* = 1 \), the approximation factor of Algorithm RSLTF is \( 1.21 \).

**Proof.** The approximation factor \( \rho \) in this case is bounded by either \( \rho_1(z) = \frac{z^3}{3z} \) or \( \max \{ \rho_2(z) = \frac{z^3}{3z} \} \). Clearly, under the condition \( 1 \leq z \leq 2 \),

\[\]
we know that \( \rho_1(z) \) is an increasing function of \( z \), while \( \rho_2(z) \) and \( \rho_3(z) \) are decreasing functions of \( z \). Therefore, the value \( \rho \) satisfies \( \rho_1(z^*) = \rho_3(z^*) = \rho \) for some \( z^* \). By solving 
\[
\frac{z^3 + 2}{3z} = \frac{4 + z}{3z},
\]
we know that \( z^* \) is \( \frac{1}{\sqrt[3]{27 + 3\sqrt{78}} + \sqrt{\frac{27 + \sqrt{78}}{3}}} \) and \( \rho \) is close to 1.21.

### 4.1.2 Analysis when \( m^* \geq 2 \)

When \( m^* \) is no less than 2, Algorithm RSLTF allocates at least two processors. Again, for notational brevity, let \( z \) be \( \sum_{r_j \in T} \frac{j}{s_j^{*+5}} \), and the lower bound of the optimal solution is \( 1.5 \beta z D \). Similar to the analysis in Section 4.1.1, what we have to do is to find an upper bound for task execution when allocating \( m^* \) or \( m^* + 1 \) processors for task execution.

We will first show how to find an upper bound \( E_{m^*} \) of energy consumption for the task partition, i.e., \( \{T_1, T_2, \ldots, T_{m^*}\} \), when \( m^* \) processors are allocated. Again, let \( z \) be \( \sum_{r_j \in T} \frac{j}{s_j^{*+5}} \).

As executing all the tasks in \( T_{k+1} \) at speed \( z_k \) on a processor is an upper bound of the derived solution, we know that \( E_{m^*} = D(m^* \beta + \sum_{m_{\ell=1}}^{m} \alpha(s_\ell)^3 z_\ell) = D(\beta (m^* + 0.5 \sum_{m_{\ell=1}}^{m} z_\ell) \)) is an upper bound. The following lemma shows the upper bound of \( \sum_{m_{\ell=1}}^{m} m^3 z_\ell \) based on Lemma 1.

**LEMMA 3.** Let \( m^1 \) be an integer, and \( Z = \sum_{m_{\ell=1}}^{m^1} z_\ell \). For any \( \ell = 1, 2, \ldots, \) \( m^1, z_\ell \geq 0 \) and \( z_\ell < 2z_\ell \) for any \( \ell \neq \ell \). Then,
\[
\sum_{m_{\ell=1}}^{m^1} m^3 z_\ell \leq 1.412 m^1 \left( \frac{Z}{m^1} \right)^3.
\]

**PROOF.** The proof is in the technical report [3].

Therefore, we know that the approximation factor \( \rho \) for this case is bounded by
\[
\rho_4(m^*, z) = \frac{2(m^* + 1) + 1.412(m^* + 1) \left( \frac{Z}{m^* + 1} \right)^3}{3z},
\]
where \( m^* \leq z < m^* + 1 \). Note that, when \( m^* = 2 \), even though the above equation stands, we will use Equation (2) for tighter analysis.

The other case by allocating \( m^* + 1 \) processors can be done by two cases. If the maximum workload among these \( m^* + 1 \) processors is more than \( s^* \), after task assignment, with a simple corollary, the approximation factor \( \rho \) is bounded by
\[
\rho_5(m^*, z) = \frac{2(m^* + 1) + 1.412(m^* + 1) \left( \frac{Z}{m^* + 1} \right)^3}{3z},
\]
where \( m^* \leq z < m^* + 1 \). Otherwise, the approximation factor \( \rho \) is bounded by
\[
\rho_6(m^*, z) = \frac{1.5 \beta D z + \beta D (m^* + 1 - z)}{1.5 \beta D z} = \frac{2(m^* + 1) + z}{3z},
\]
while the denominator is the energy consumption by executing at the critical speed without turning the processor to the dormant mode.

Therefore, similar to Equation (1), when \( m^* \geq 2 \), the approximation factor \( \rho \) is
\[
\rho = \max_{2 \leq m^* \leq z < m^* + 1} \min\{\rho_4(m^*, z), \max\{\rho_5(m^*, z), \rho_6(m^*, z)\}\},
\]
where, more precisely, \( \rho_4(m^*, z) \) can be replaced by \( \rho_2(z) = \frac{z^3 + 4}{3z^3} \) when \( m^* = 2 \). When \( m^* \geq 6 \), it is not difficult to see that
\[
\rho_6(m^*, z) < \rho_5(m^*, z) < \rho_4(m^*, z),
\]
for any \( 6 \leq m^* \leq z < m^* + 1 \). Hence, \( \rho_5(m^*, z) \) dominates the approximation factor when \( m^* \geq 6 \). Because \( \rho_5(m^*, z) \) is a convex function of \( z \) for a fixed \( m^* \), we know that either \( \rho_5(m^*, m^*) \) or \( \rho_5(m^*, m^* + 1) \) is the maximal value. Moreover, when \( m^* \geq 6 \), we have
\[
\rho_5(m^*, z) \leq \rho_5(m^*, m^* + 1) \leq 1.138.
\]

Figure 5 illustrates the corresponding functions \( \rho_2(z), \rho_4(z), \rho_5(z) \), and \( \rho_6(z) \) when \( m^* = 2, 3, 4, 5 \). As shown in Figure 5(a), the intersection of \( \rho_2(z) \) and \( \rho_5(z) \) is \( 2.3553 \) is the bound when \( m^* = 2 \). For the case that \( m^* = 3 \) (\( m^* = 4 \), respectively), the intersection of \( \rho_5(z^*) \) and \( \rho_6(m^*, z) \) when \( z^* \) is \( 3.2154 \) (\( 4.1051 \), respectively) bounds the approximation factor. Moreover, when \( m^* = 5 \), the approximation factor is bounded by \( \rho_5(6, 6) = 1.138 \).

By concluding all the above cases when \( m^* = 0, 1, 2, 3, 4, 5 \), and \( m^* \geq 6 \), we can reach the following concluding theorem.

**THEOREM 1.** When \( s_{min} = 0 \), the approximation factor of Algorithm RSLTF is 1.21.

**PROOF.** By Lemma 2 and the above discussions, the approximation factor \( \rho \) is 1 when \( m^* = 0 \), and there is no more than 1.21 when \( m^* = 1, 1.83 \text{ when } m^* = 2, 1.163 \text{ when } m^* = 3, 1.146 \text{ when } m^* = 4, \text{ and } 1.138 \text{ when } m^* \geq 5 \). Therefore, we reach the conclusion of the approximation factor \( \rho \leq 1.21 \).

### 4.2 Extensions to periodic real-time tasks

For periodic real-time tasks, the extension of Algorithm RSLTF is simple by changing all the operations in execution cycles \( c_j \) to the cycle utilization \( \frac{c_j}{T_j} \). That is, Step 3 in Algorithm 1 is revised to a non-increasing order of \( \frac{c_j}{T_j} \). Step 2 in Algorithm 1 is revised as \( m^* \leftarrow \lceil \sum_{r_j \in T} \frac{c_j}{T_j} \rceil \), and Step 7 in Algorithm 1 is revised as \( C_{1, s_1} \leftarrow C_{1, s_1} + \frac{c_j}{T_j}, C_{2, s_2} \leftarrow C_{2, s_2} + \frac{c_j}{T_j} \). The only problem is the
With similar analysis in Section 4.1, the following corollary holds:

Corollary 1. When \( s_{\text{min}} > 0 \), the approximation factor of Algorithm RSLTF is 1.43.

5. PERFORMANCE EVALUATION

In this section, we provide performance evaluation on the energy consumption of the proposed algorithm. For comparison, we also evaluate the proposed algorithms proposed in [1]. Moreover, the following algorithms are evaluated for comparisons:

- Algorithm RSLTF-CRITICAL (or P-RSLTF-CRITICAL for periodic real-time tasks) applies the task partition algorithm in this paper by taking the critical speed as the lower bound for task execution.
- Algorithm LALTF-FIRST-FIT is Algorithm LA+LTF in [1] by applying the first-fit strategy for task remapping.
- Algorithm LALTF-WORST-FIT is Algorithm LA+LTF in [1] by applying the worst-fit strategy for task remapping.

For Algorithm LALTF, we also evaluate the best-fit and the last-fit strategies for task remapping, but, their results are either close to those of the first-fit or the worst-fit strategy. For the clarity of figures, we will not include their results. For all evaluated algorithms, we also apply the existing procrastination approach [8].

Workload Parameters and Performance Metrics. We evaluate the energy consumption of the solutions by using the power consumption function of Intel XScale, in which \( P(s) \) is modeled by \( 1.52 (\frac{s}{\text{GHz}})^{3} + 0.08 \text{ Watt} \) [2,12]. The available speeds are in the range of \([s_{\text{min}}, 1 \text{ GHz}]\), where \( s_{\text{min}} \) is a specified input in our settings. The energy overhead \( E_{\text{sw}} \) of switching is in the range of \([0.2, 1.5] \text{ mJoule} \), and the timing overhead \( t_{\text{sw}} \) of switching is assumed to be 0.1 msec.

We use synthetic real-time task sets for performance evaluation. For each task \( \tau_{j} \), the execution time at 1 GHz is a random variable uniformly distributed in the range of \([0.01, 0.297] \mu \text{s} \), where \( p_{j} \) is 30 msec for the setting of frame-based real-time tasks, and is a random variable in the range of \([10, 100] \mu \text{s} \) for the setting of periodic real-time tasks. Because Algorithm RSLTF applies Algorithm LALTF as a subroutine when \( \sum_{j \in \tau} \frac{s_{\text{sw}}}{\text{GHz}} \) is more than the critical speed, to show the effectiveness of Algorithm RSLTF, the number of processors \( m \) is set as \( 2 \left[ \sum_{j \in \tau} \frac{s_{\text{sw}}}{\text{GHz}} \right] + 1 \) for each task set, where \( \sum_{j \in \tau} \frac{s_{\text{sw}}}{\text{GHz}} > 1 \).

We evaluate three different settings by (1) changing the number of tasks from 4 to 32 with \( s_{\text{min}} = 0 \) and \( E_{\text{sw}} = 1 \text{ mJoule} \), (2) changing the energy overhead \( E_{\text{sw}} \) of switching from 0.2 mJoule to 1.5 mJoule for 20 tasks with \( s_{\text{min}} = 0 \), and (3) changing the minimum available speed \( s_{\text{min}} \) from 0.2 GHz to 0.25GHz for 20 tasks with \( E_{\text{sw}} = 1 \text{ mJoule} \). Because procrastination might be performed, for an input instance, we evaluate the energy consumption of frame-based real-time tasks for 3 sec and report the average energy consumption in 30 msec as the result. For periodic real-time tasks, the energy consumption is evaluated in 3600 sec.

For each configuration, we perform 512 independent simulations, and report the average energy consumption. Moreover, to show the derived solutions of Algorithm RSLTF are quite close to the optimal solutions, we also report the energy consumption by executing all the tasks at the critical speed as the Lower Bound of the optimal energy consumption by ignoring the energy overhead \( E_{\text{sw}} \). The normalized energy of an algorithm for an input instance is the energy consumption of the derived solution divided by the above lower bound. For each configuration, the average normalized energy is reported.

Evaluation Results. Figure 6 shows the results for frame-based real-time tasks with \( s_{\text{min}} = 0 \) and \( E_{\text{sw}} = 1 \text{ mJoule} \), where Figure 6(a) is the average energy consumption per 30 msec, and Figure 6(b) is the average normalized energy consumption. Clearly, when the number of tasks increases, the resulting energy consumption increases. However, in terms of normalized energy consumption, all the evaluated algorithms are worse when the number of tasks is small. Similar to the argument of non-optimality of critical speeds in Section 3, the comparison of Algorithms RSLTF and RSLTF-CRITICAL in Figure 6(b) shows that taking the critical speed as the lower bound for task execution could be worse. This is because most processors assigned with tasks by applying Algorithm RSLTF for partitioning tasks are almost fully-utilized for task execution. Procrastination does not help too much, and, most of time, making the resulting schedules worse. The improvement of Algorithm RSLTF, compared to Algorithm LALTF, is about 8 ~ 13.5%.

Figure 7 illustrates the average energy consumption in 30 msec
for 20 frame-based real-time tasks with $s_{\text{min}} = 0$ for different energy overheads $E_{\text{sw}}$ of switching. Again, Algorithm \textsc{RSLTF-CRITICAL} does not help reduce the energy with the same reason in Figure 6. Because, most of time, Algorithm \textsc{LALTF-FIRST-FIT} and Algorithm \textsc{LALTF-WORST-FIT} would use more processors for task execution, when the energy overhead $E_{\text{sw}}$ of switching is lower, they have better results. The improvement of Algorithm \textsc{RSLTF}, compared to Algorithm \textsc{LALTF}, is about $3 \sim 8\%$. Figure 8 presents the results for 20 frame-based real-time tasks with $E_{\text{sw}} = 1 \text{ mJoule}$, in which Algorithm \textsc{RSLTF} is better when $s_{\text{min}}$ is higher.

Figure 9 shows the normalized energy consumption for periodic real-time tasks. The results are similar to frame-based real-time tasks. Since the optimal uniprocessor schedules can be determined easily for frame-based real-time tasks, in general, the normalized energy consumption for periodic real-time tasks is worse than that for frame-based real-time tasks. As shown in the simulations, Algorithm \textsc{RSLTF} outperforms the family of Algorithm \textsc{LALTF} in both average cases and worst cases, while the improvement in average cases could be at most 15\%.

6. CONCLUSION

This paper explores energy-efficient scheduling in leakage-aware DVS systems. Even though the critical speed is optimal for job/task execution, we show that it might not be good enough to stick with the critical speed by taking it as the lower-bounded speed for execution or by treating it as the upper-bounded speed for task mapping in multiprocessor systems. Moreover, we also show that load balancing might not be good since assigning tasks on a light-loaded processor might waste energy to turn on/off the processor. The above observation motivates the research in this paper to improve existing scheduling algorithms for minimizing the energy consumption in homogeneous multiprocessor systems. This paper presents an efficient algorithm, which allocates the processors based on the critical speed and performs task assignment for load balancing without considering the critical speed. When multiple processors are necessary for energy minimization, the proposed algorithm is proved to derive solutions with at most 1.21 (1.43, respectively) times of the energy consumption of the corresponding optimal solutions when the minimum available speed is 0 (not 0, respectively). Experimental results reveal that the proposed algorithm is more effective than existing algorithms also in average cases.

References


