Leakage-Aware Scheduling for Embedded Real-Time Systems with (m, k)-Constraints

Abstract

In this paper, we study the problem of reducing both the dynamic and leakage energy consumption for real-time systems with (m, k)-constraints, which require that at least m out of any k consecutive jobs of a task meet their deadlines. Two energy efficient scheduling approaches incorporating both dynamic voltage scheduling (DVS) and dynamic power down (DPD) are proposed in this paper. The first one statically determines the mandatory jobs that need to meet their deadlines in order to satisfy the (m, k)-constraints, and the second one does so dynamically. The simulation results demonstrate that, with more accurate workload estimation, our proposed techniques outperformed previous research in both overall and idle energy reduction while providing the (m, k)-guarantee.

Keywords leakage, power-aware scheduling, (m, k)-guarantee, real-time system

1 Introduction

As transistor density continues to grow, the power/energy conservation problem becomes more and more critical in the design of pervasive embedded real-time systems. For CMOS circuits, the power consumption comes from the dynamic power consumption (mainly due to switching activities) and the static power consumption (mainly due to leakage current). As VLSI technology continues its evolution toward the submicron and nanoscale era, the rapidly elevating leakage power dissipation is becoming too prominent to be ignored [19].

The *dynamic voltage scaling* (DVS) strategy (e.g., [3, 22, 42]), *i.e.*, by dynamically changing the supply voltage as well as the working frequency, has long be recognized as an effective method to reduce the dynamic energy. However, as the leakage power continues to increase, the energy saving achievable via DVS alone is becoming severely limited. This is because DVS prolongs the active period of the processor and thus can increase the total leakage energy consumption [21]. The *dynamic power down* (DPD) strategy, on the other hand, dynamically turns the system on and off and is thus an effective way to control the leakage energy consumption at the system level. Therefore, to obtain the maximal reduction in the overall energy consumption, some researchers [21, 26, 7, 8] have proposed to combine DVS and DPD to reduce both the dynamic power and leakage power simultaneously. Most of the approaches have targeted the hard real-time systems, *i.e.*, the systems requiring that all the task instances meet their deadlines.

While the hard real-time model is the simplest model for real-time systems, few real-time applications are truly *hard* real-time. For example, many practical real-time applications such as multimedia processing or real-time communication systems exhibit more complicated characteristics that can only be captured with more complex requirements, generally called the *Quality of Service (QoS) requirements*. For example, some applications may have soft deadlines where tasks that do not finish by their deadlines can still be completed with a reduced value [28] or they can simply be dropped without compromising the desired QoS levels. Energy reduction under QoS requirement falls within the framework of more general resource management/scheduling, such as the QoS-based

Resource Allocation Model (Q-RAM) [38]. A key to the success is the ability to integrate the QoS requirements into resource management/scheduling decisions in such a way that the overall "benefit" of the system is optimized. The techniques based on the traditional hard real-time systems become inefficient or inadequate when QoS requirements are imposed on the systems.

To quantify the QoS requirements, some statistic information such as the average deadline miss rate is commonly used. Although the statistical deadline miss rate can ensure the quality of service in a probabilistic manner, this metric can be problematic for some real-time applications. For example, for certain real-time systems, when the deadline misses happened to some tasks, the information carried by those tasks can be estimated in a reasonable accuracy using techniques such as interpolation. However, even a very low overall miss rate tolerance cannot prevent a large number of deadline misses from occurring in such a short period of time that the data cannot be successfully reconstructed. To avoid possible severe consequences, one can always treat the system as a hard real-time system. The problem, however, is that when the system is overloaded (this is especially common when energy conservation techniques such as DVS are applied to the system, as reducing the processor voltage/frequcies will increase the execution times of the tasks and thus easily cause the systems to be "overloaded"), the approaches for hard real-time systems are no longer valid as deadline missing will become inevitable in such kind of scenarios. As a result, critical information that cannot be reconstructed may be lost, thus the service quality can be severely degraded from the user's perspective.

To provide deterministic QoS to the real-time system, the system should not only support the overall guarantee of the QoS statistically, but also be able to provide a lower bounded, predictable level of QoS locally. The (m,k)-model, proposed by Hamdaoui *et al.* [14], can serve well for this purpose. According to this model, a repetitive task of the system is associated with an $(m,k)(0 < m \leq k)$ constraint requiring that m out of any k consecutive job instances of the task meet their deadlines. A *dynamic failure* occurs, which implies that the temporal QoS constraint is violated and the scheduler is thus considered failed, if within any sliding window of k consecutive jobs less than m job instances meet their deadlines. Due to its intuitiveness and capability of capturing the deterministic QoS requirements, (m,k)-model has been widely studied, e.g., [5, 35, 14, 39, 15].

In this paper, we study the problem of reducing the overall energy consumption for real-time systems under the $(m,k)-$constraints. In order to guarantee the $(m,k)-$constraints be satisfied all the time, a key aspect of this problem is to judiciously partition the jobs into *mandatory* jobs and *optional* jobs such that as far as all the mandatory jobs can meet their deadlines, the (m,k)-constraints can be ensured. We proposed two approaches to address this problem. In the first approach, the mandatory/optional partition is conducted statically. The mandatory jobs are carefully procrastinated with the purpose of merging the idle intervals so that the processor can be shut down effectively. In the second approach, we dynamically adjust the mandatory/optional job partitioning to accommodate the dynamic nature of real-time embedded systems. In particular, a look-ahead strategy is proposed that can more accurately estimate the workload and therefore make better decisions in the mandatory/optional job partitioning. Moreover, considering the variations in job execution times at runtime, we also developed a new dynamic reclaiming approach with looking forward technique that can incorporate some previous work [4, 22] to improve the energy saving performance. Different from the previous works [20, 26, 36, 7, 9, 8] that use the so called "critical speed" as the lower bound to optimize the overall energy, we found that executing the jobs with speed lower than the critical speed can sometimes be more beneficial in overall energy savings. With a practical processor model and technology parameters [17], our experiment results show that our approach significantly outperformed the existing research in energy saving performance while providing the (m,k)-guarantee.

The rest of the paper is organized as follows. Section 2 discusses about the related work. Section 3 introduces the system models and some preliminaries. Section 4 introduces our static approach. Section 5 introduces our dynamic approach. The effectiveness and energy efficiency of our approaches are evaluated in section 7. In section 8, we offer the conclusions and future work.

2 Related work

Many scheduling techniques (e.g.[42, 18, 4, 33, 34, 45, 46]) have been proposed to reduce the power/energy consumption for real-time embedded systems. Most of them have focused on reducing the dynamic power/energy consumption. However, as VLSI technology marching towards deep submicron and nanoscale circuits operating at multi-GHz frequencies, the rapidly elevated leakage power dissipation will soon become comparable to, if not exceeding, the dynamic power consumption [19]. To balance the dynamic power and leakage power, the concept of *critical speed* was proposed, which is the speed that can minimize the active energy of a job during its execution. Based on it, many scheduling techniques [20, 26, 36, 7, 9, 8] have been proposed to reduce dynamic power and leakage power simultaneously. And most of them use the critical speed as the lower bound for speed reduction. Since the critical speed may require the processor to run at *higher-than-necessary* speeds and can result in a large number of idle intervals, more advanced techniques [21, 26, 36, 7] based on procrastination of jobs have been proposed to merge/prolong the idle intervals. However, is it necessary to always keep the job speed above the critical speed? In this paper, we show that sometimes it can be more beneficial to execute the jobs with speed lower than the critical speed in saving the overall energy consumption.

Some researchers [9, 10, 7, 47] have also noticed the possibility of improving energy efficiency by reducing the job speed below the critical speed. However, few in-depth efforts were made to explore the advantages of breaking-through this lower bound for general task sets under the context of QoS constraints. In [23, 44, 12], advanced techniques were provided to minimize the system-wide energy consumption by exploring the interplay between DVS and DPD for real time systems containing a core processor and multiple devices. Their approaches target the so called "frame-based" system which is a very special case of real-time applications that all tasks have the same deadlines and periods. And the impact of task procrastination on speed selection is not considered, either.

Some previous researches have also been reported to reduce the energy for real time systems with (m,k) requirements. In [31], a hybrid approach was introduced to reduce the dynamic energy consumption for real time systems with (m,k)-guarantee. With QoS requirements formulated as a tolerable statistical deadline miss rate, Hua *et al.* [16] introduced several techniques to reduce energy by exploiting processor slack time due to the missed deadlines. In [2], Alenawy *et. al.* proposed an approach to minimize the number of dynamic failures for (m,k)-firm systems with fixed energy budget constraint. In [25], a dynamic approach is proposed to minimize the energy consumption for *dual-voltage-mode* weakly hard real-time systems. In [32], Niu *et al.* proposed an approach to reduce energy for weakly hard real-time systems with peripheral devices. All of these techniques targeted dynamic power reduction and none of them considered reducing the dynamic and leakage energy consumption simultaneously.

3 Preliminary

In this section, we first give the system model and power model. Then we introduce some concepts and observations important to our research in this paper.

3.1 System model

The real-time system considered in this paper contains n independent periodic tasks, $\mathcal{T} = \{\tau_0, \tau_1, \cdots, \tau_{n-1}\}$, scheduled according to the earliest deadline first (EDF) policy [27]. Each task contains an infinite sequence of periodically arriving instances called *jobs*. Task τ_i is characterized using five parameters, *i.e.*, $(T_i, D_i, C_i, m_i, k_i)$. $T_i, D_i (D_i \leq T_i)$, and C_i represent the period, the deadline and the worst case execution time for τ_i, respectively. The QoS requirement for τ_i is represented by a pair of integers, i.e., (m_i,k_i) $(0 < m_i \leq k_i)$, which require that, among any k_i consecutive jobs of τ_i, at least m_i jobs must meet their deadlines. The j^{th} job of task τ_i is represented with J_{ij} and its arrival time, actual execution time and absolute deadline are represented by r_{ij}, c_{ij} and d_{ij}. It is not hard to see that the arrival times r_{ij} of J_{ij} can be computed as $r_{ij} = (j-1) * T_i$ and its absolute deadline d_{ij} can be computed as $d_{ij} = r_{ij} + D_i$. Also we assume once job J_{ij} arrives, its actual execution time c_{ij} can be available.

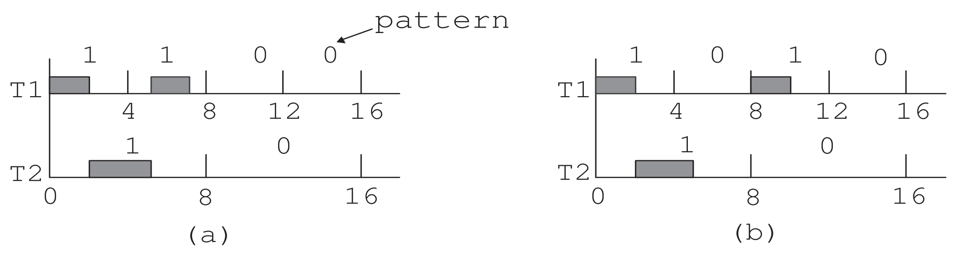


Figure 1. (a) The schedule of a task set ($\tau_1 = (4,4,2,2,4)$; $\tau_2 = (8,8,3,1,2)$) partitioned with R-Pattern; (b) The schedule of the same task set partitioned with E-Pattern.

3.2 The power model

In a CMOS circuit, the power consumption includes both dynamic and static components during its active operation. The dynamic power consumption (P_{dyn}) mainly consists of the switching power for charging and discharging the load capacitance, which can be represented [6] as

$$P_{dyn} = \alpha C_L V^2 f, \tag{1}$$

where α is the switching activity, C_L is the load capacitance, V is the supply voltage, and f is the system clock frequency. The static power (P_{leak}) can be expressed [29] as

$$P_{leak} = I_{leak} V, \tag{2}$$

where I_{leak} is the leakage current which consists of both the subthreshold leakage current and the reverse bias junction current in the CMOS circuit. Leakage current increases rapidly with the scaling of the devices and becomes particularly significant with the reduction of the threshold voltage. Therefore, the leakage power consumption is becoming a major part of the the active power consumption ($P_{act} = P_{dyn} + P_{leak}$) in future CMOS circuits with low supply voltage and high transistor density.

To reduce the overall energy consumption, one common method is to use the so called *critical speed* [20] (denoted as s_{crit}) to minimize the active energy of executing a job. From [20], using a processor speed higher or lower than the critical speed will consume more active energy than the one using the critical speed to complete the same workload.

The DVS processor used in our system can operate at a finite set of discrete supply voltage levels $\mathcal{V} = \{V_1, ..., V_{max}\}$, each with an associated speed. To simplify the discussion, we normalize the processor speeds to S_{max}, the speed corresponding to V_{max}, which results in $\mathcal{S} = \{S_1, ..., 1\}$. We assume that C_i is the worst case execution time for task τ_i in the highest voltage mode. Therefore, if τ_i is executed under speed S_j, the worst case execution time for τ_i becomes $\frac{C_i}{S_j}$.

The processor can be in one of the three states: *active*, *idle* and *sleeping* states. When the processor is idle, the major portion of the power consumption comes from the leakage which increases rapidly with the dramatic increasing of the leakage power consumption. Shutting-down strategy, *i.e.*, put the processor into its sleeping state, can greatly reduce the leakage energy. However, it has to pay extra energy and timing overhead to shut down and later wake up the processor. Assume that the power consumptions of a processor in its idle state and sleeping state are P_{idle} and P_{sleep}, respectively, and the energy overhead and the timing overhead of shutdown/wakeup is E_o and t_o. Then the processor can be shut down with positive energy gains only when the length of the idle interval is larger than $T_{th} = \max(\frac{E_o}{P_{idle} - P_{sleep}}, t_o)$. We call T_{th} as the *shut down threshold interval.*

3.3 Meeting the (m,k)-constraints

A key problem for meeting the (m,k)-constraints is to judiciously partition the jobs into *mandatory* jobs and *optional* jobs [35]. The partition can be done statically or dynamically. Two well-known partition strategies proposed are the *deeply-red pattern* (or R-pattern) and *evenly distributed pattern* (or E-pattern) [31]. One example of mandatory/optional job partitioning with R-pattern and E-pattern for a given task set is shown in Figure 1.

The most significant advantage of applying static patterns is that they enable the application of theoretic real-time techniques to analyze system feasibility. The problem, however, is its poor adaptivity in dealing with the run-time variations, which is inherent in many real-time applications. As shown, the R-pattern assigns the first m jobs as mandatory and tends to generate longer idle intervals, which are more beneficial for powering down the system dynamically. It has been proved that as long as all mandatory jobs selected from R-pattern can meet their deadlines, the mandatory jobs selected from any other (m,k)-pattern can also meet their deadlines [31]. The mandatory/optional partitioning according to E-pattern has the property that it helps to spread out the mandatory jobs evenly in each task along the time. Interested readers can refer to [31] for more technical details about the R-pattern and E-pattern. Based on the mandatory/optional job partitioning, in the following sections, we introduce our static and dynamic approaches to reduce the overall energy consumption while ensuring the (m,k)-guarantee.

4 The static approach

In our static approach, we first decide the mandatory job set based on E-patterns. As shown in [31], with the E-patterns, a minimal number of mandatory jobs are determined. Therefore, no energy is wasted to execute jobs that are nonessential for the (m,k)-guarantee. Moreover, since E-pattern in general helps to spread the workload evenly, it helps to better reduce the processor speed.

Once the mandatory jobs are determined with the E-pattern, we could scale down the processor speed for all tasks with the feasibility condition introduced in [31]. Since executing the job with the critical speed can minimize the active energy consumption when completing the same job workload, here we try to scale the processor speeds for all tasks as low as (but not below) the critical speed s_{crit} and use them as the predetermined speeds for the mandatory jobs. Note that, even after scaling down, there still exist a large number of idle intervals due to the potentially *higher than necessary* speed assignment. Some of them will be too short for the processor to shut down. Even if some idle intervals are longer than the shut down threshold T_{th}, too many idle intervals will also increase the energy consumption significantly due to the overheads of frequently shutting down and waking up the processor. In our static approach we try to reduce this part of energy by procrastinating the execution of some mandatory jobs.

4.1 Mandatory jobs procrastination

Procrastinating the execution of mandatory jobs can help extend or merge the idle intervals [21, 26]. However, it might potentially cause some other mandatory jobs with lower priorities to miss their deadlines. In addition, delaying the mandatory jobs not sufficiently might generate new scattered idle intervals that can not be shut down. The problem then becomes how to determine the maximal delay to merge the idle intervals without causing any mandatory jobs to miss their deadlines. In the following, we introduce two sufficient conditions to help identify the delays for mandatory jobs. Before that, we first introduce the following definitions.

Definition 1 ***(Level-i busy period)*** *Given task set $\mathcal{T} = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$ with tasks ordered by increasing value of D_i, the* **level-i busy period** *is defined to be the busy period in which only mandatory instances of tasks with relative deadlines less than or equal to D_i executed.*

Note that here we borrow the concept of *level-i* busy period from [24] to represent the busy period under EDF scheme. And it is shown in [13] that the length of the first *level-i* busy period is independent of the scheduling algorithm and must be the smallest t such that the accumulated *level-i* workload within the interval $[0,t]$ equals t.

Moreover, we have the following Theorem.

Theorem 1 *Given task set* $\mathcal{T} = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$ *with tasks ordered by increasing value of* D_i*, let* $\mathcal{E}$ *be the mandatory jobs selected from* $\mathcal{T}$ *based on their E-patterns, and let L be the ending point of the first busy period when executing the mandatory jobs. Let* B_i *be the blocking factor for task* τ_i*,* i.e.*, the longest time that a job of* τ_i *can be blocked by other lower priority jobs.* $\mathcal{E}$ *is schedulable* if

$$\sum_{D_i \le t} (\lceil \frac{m_i}{k_i} \lfloor \frac{t + T_i - D_i}{T_i} \rfloor \rceil) C_i + B_{l(t)} \le t \tag{3}$$

for all $t \le L$*,* $t = \lfloor p \frac{k_i}{m_i} \rfloor T_i + D_i, p \in Z, i = 0, ..., n-1$ *and* $l(t) = \max\{l | D_l \le t\}$.

(The proof is provided in the Appendix part A.)

Based on Theorem 1, given a real-time system $\mathcal{T}$ with tasks ordered by increasing value of D_i, the maximal tolerable blocking factor B_i for each task τ_i under E-pattern satisfying Equation (3) can be computed by: (Similar theorem can be established based on R-patten as well and the maximal tolerable blocking factor under R-pattern can be found in the same way.)

$$B_i = \min\{t - \sum_{j \le i} (\lceil \frac{m_i}{k_i} \lfloor \frac{t + T_i - D_i}{T_i} \rfloor \rceil) C_i\} \tag{4}$$

for all $t = \lfloor p \frac{k_j}{m_j} \rfloor T_j + D_j, p \in Z, j = 0, ..., i$ such that $D_i \le t \le \lfloor \frac{L_i}{T_j} \rfloor T_j + D_j$, where L_i is the length of the first *level-i* busy period in which only mandatory instances of tasks with deadlines less than or equal to D_i execute.

It is not hard to see that the blocking factor B_i computed by Equation (4) can satisfy Equation (3). Moreover, although the definition of B_i in Theorem 1 is based on Stack Resource Policy [41] and the Dynamic Priority Ceiling Protocols [11], it can also be used in our system model to implement procrastination on the mandatory jobs, which is stated in the following theorem (The proof is provided in the Appendix part B):

Theorem 2 *Given task set* $\mathcal{T} = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$*, let* $\mathcal{E}$ *be the mandatory jobs selected from* $\mathcal{T}$ *based on their E-patterns. Let* B_i *be the blocking factor for task* τ_i *computed by Equation (4). For any mandatory job* J_i *of* τ_i *in* $\mathcal{E}$*, if it is procrastinated by no more than* B_i *time units after its arrival, no mandatory job in* $\mathcal{E}$ *will miss its deadline.*

With Theorem 2, we can formulate the first sufficient condition as follows.

Theorem 3 *Let* $\mathcal{M}$ *be the mandatory job set based on E-pattern or R-pattern, and let* B_i *be the corresponding blocking factor for task* τ_i *computed by Equation (4). Let the current time be t, and let the coming mandatory job set (i.e. with arrival time* r_i *no earlier than t) be* $\mathcal{J}'$*. If the execution of* $\mathcal{J}'$ *is delayed to*

$$T_{LS}(\mathcal{J}') = \min_{J_i \in \mathcal{J}'} (r_i + B_i), i = 0, 1, ..., n-1, \tag{5}$$

no mandatory job in $\mathcal{M}$ *will miss its deadline.*

The proof can be done based on Theorem 2 and is provided in Appendix part C.

Theorem 3 provides one method to calculate the latest time that the upcoming mandatory jobs can be delayed without incurring any deadline misses for the mandatory jobs. Moreover, given that B_i is available, we can also develop another method to identify the maximal delay for a mandatory job set, as stated in Theorem 4 (The proof is provided in Appendix part D).

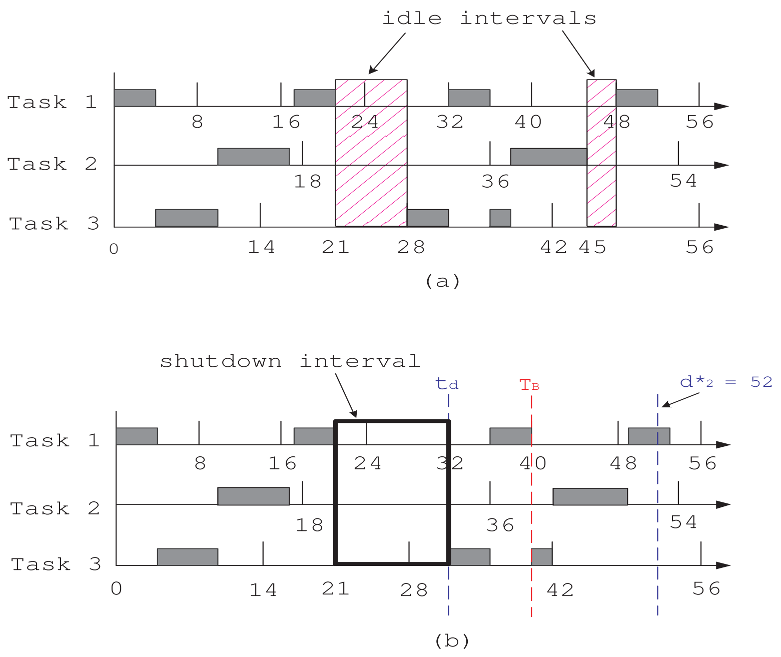


Figure 2. (a) The schedule of a task set ($\tau_1 = (8,8,4,2,4)$; $\tau_2 = (18,18,7,2,4)$; $\tau_2 = (14,14,6,1,2)$) with mandatory jobs determined with E-Pattern; (b) With mandatory jobs procrastination, the same task set merged the idle intervals effectively.

Theorem 4 *Let $\mathcal{M}$ be the mandatory job set based on E-pattern or R-pattern, and let B_i be the corresponding blocking factor for task τ_i computed by Equation (4). Let the processor speed for each task τ_k be s_k. Assuming the current time is $t = t_0$, and let the coming mandatory job set (i.e. with arrive time r_i later than t_0) be $\mathcal{J}'$. Let the earliest deadline for the mandatory jobs in $\mathcal{J}'$ be T_B. Then no mandatory job in $\mathcal{J}'$ will miss its deadline if the execution of all mandatory jobs in $\mathcal{J}'$ is delayed to $T_{LS}(\mathcal{J}')$, where*

$$T_{LS}(\mathcal{J}') = \min_{J_i \in \mathcal{J}_s} (d_i^* - \sum_{J_k \in hp(J_i)} \frac{c_k}{s_k}), \tag{6}$$

where $\mathcal{J}_s$ consists of mandatory jobs from $\mathcal{J}'$ with arrival times earlier than T_B but later than t_0, $hp(J_i)$ are the jobs with equal or higher priorities than J_i and

$$d_i^* = \min_p (d_i, r_p + B_p), \forall J_p \in \mathcal{J}', J_p \notin \mathcal{J}_s \text{ and } d_p > d_i. \tag{7}$$

The fundamental difference between our technique in Theorem 4 and the one in [26] is the way that the effective deadline d_i^* is defined. From Equation (7), the effective deadline for a mandatory job is prolonged with the blocking factor B_p of the next mandatory jobs. This in turn will allow the mandatory jobs to be delayed further. While it might not be always energy efficient to delay the mandatory jobs to the maximal extent, having larger delay interval can provide us more flexibility in determining the most energy efficient speeds for jobs online. Note that since both Theorem 3 and Theorem 4 are sufficient conditions, the larger one from equation (15) and (6) can be used as the latest starting time t_d for the upcoming mandatory jobs. Finally, since our sufficient conditions in Theorem 3 and Theorem 4 do not depend on whether the system is idle at time t or not, so they can be used to compute the maximal delay for the upcoming mandatory job set not only when the processor is idle, but also when the processor is busy at time t, which will be very useful for our dynamic approach introduced in Section 5.

4.2 One example

As an example, consider a the task set of three tasks ($\tau_1 = (8,8,4,2,4)$; $\tau_2 = (18,18,7,2,4)$; $\tau_2 = (14,14,6,1,2)$). Assume all tasks are executed with the highest speed, *i.e.*, $s_1 = s_2 = s_3 = 1$. The task schedule for mandatory jobs determined with E-pattern is shown in Figure 2(a). If we assume the shutdown threshold $T_{th} = 10$, as seen in Figure 2(a), 2 idle intervals were generated at $t = 21$ and $t = 45$ respectively and neither of them can be shut down.

However, if we apply Theorem 3 and Theorem 4 at $t = 21$, as shown in Figure 2(b), the future mandatory jobs can be delayed to $t_d = 32$ and as a result the two short idle intervals can be merged into a longer one which can be shut down safely. (Here according to Equation (4), $B_1 = 4$, $B_2 = 1$, and $B_3 = 4$. So at $t = 21$, by applying Theorem 3, the future mandatory jobs can be delayed to $t = 32$. Note that, when applying Theorem 4, task τ_2 alone can be delayed further because at $t = 21$, $T_B = 40$ and from Equation (7), the effective deadline for τ_2 is $d_2^* = 52$, as shown in Figure 2(b). Therefore the effective starting time for τ_2 can be up to $t = 45$. However, since in Theorem 4, according to Equation (6), the latest starting time for the upcoming mandatory jobs, *i.e.*, $T_{LS}(\mathcal{J}')$, is constrained by all future mandatory jobs arriving before T_B, $T_{LS}(\mathcal{J}')$ should be chosen such that the deadlines of all upcoming mandatory jobs could be ensured. Consequently the latest starting time computed with $d_2^* = 52$ is still $t = 32$.)

4.3 The algorithm for static approach

Based on Theorem 3 and Theorem 4, the algorithm for our static approach can be formulated as followed.

Algorithm 1 The algorithm for static approach. (Algorithm *LKST*)

```
 1: Input: The current time $t_{cur}$.
 2: $t_d$ = the latest starting time for the upcoming mandatory jobs according to Theorem 3 and Theorem 4;
 3: if the processor is idle then
 4:     if $(t_d - t_{cur}) > T_{th}$ then
 5:         Shut down the processor and set up the wake up timer to be $(t_d - t_{cur})$;
 6:     end if
 7: else if $J_i$ is the only job in the ready queue then
 8:     $s_i' = max\{\frac{c_i s_i}{(\min\{t_d, d_i\} - t_{cur})}, s_{crit}\}$;
 9:     $f_i = t_{cur} + \frac{c_i s_i}{s_i'}$; // scale $s_i$ to be no less than $s_{crit}$
10:     if $(t_d - f_i) > T_{th}$ then
11:         Execute $J_i$ with $s_i'$ and upon completion of $J_i$ shut down the processor and set up the wake up timer to be
            $(t_d - f_i)$;
12:     else
13:         $s_i' = \frac{c_i s_i}{(\min\{t_d, d_i\} - t_{cur})}$;// scale $s_i$ as low as possible
14:     end if
15: else
16:     Run jobs in the ready queue with their predetermined speeds according to EDF;
17: end if
```

As shown in Algorithm 1, a timer is maintained to keep track of time and wake up the processor after a specified time interval. At current time t_{cur}, if the processor begins to idle, we compute the latest starting time t_d for the upcoming mandatory jobs and shut down the processor if $(t_d - t_{cur}) > T_{th}$. Otherwise we will schedule the mandatory jobs with their predetermined speeds according to EDF scheme. One special case for the latter one is whenever there is only one mandatory job in the ready queue, by delaying the upcoming mandatory jobs to t_d, we can always use the available time to scale the speed of J_i as low as s_{crit} first. If there is still extra idle time before

t_d and it is larger than T_{th}, we can shut down the processor during this idle interval (Line 6 - Line 9). Otherwise instead of letting the processor idle during this interval, we can utilize the remaining time to reduce the speed of J_i as low as possible to achieve further energy savings (Line 11).

Note that the difference between our static approach and the approach in [21] is that the approach in [21] is based on individual job procrastination upon its arrival, while our approach is based on computing the latest starting time for the whole upcoming mandatory job set at any time. As stated in the proof procedure of Theorem 2, in our approach, the procrastination time for any of the upcoming mandatory job J_i is not necessarily B_i time units in practice even if the processor is idle at r_i. On the contrary, in the approach from [20], each job J_i is always procrastinated by Z_i time units only if the processor is already in the sleeping state at r_i (interested readers can refer to [20] for more details). Consequently, to guarantee the schedulability of the task set, one additional constraint has to be imposed on the procrastination time Z_i for each task τ_i, *i.e*, $\forall k \leq i, Z_k \leq Z_i$. As a result, the procrastination time Z_i achievable in the approach from [20] is rather pessimistic. Generally our approach can predict the idle interval length and shut down the processor more efficiently than the approach in [20]. Moreover, when $(m,k)-$constraits are imposed to the system, the utilization based condition in [20] to compute the procrastination time Z_i is not applicable any more while our approach can be applied with more general system model.

5 The dynamic approach

The advantages of the static approach introduced above are that the mandatory job set is the minimal, and the mandatory jobs are evenly distributed with respect to each task. However, even though the mandatory jobs for each task are evenly distributed, the overall mandatory workload are not necessarily evenly distributed. Moreover, since E-pattern tends to separate mandatory jobs away from each other, it may generate a large number of idle intervals. It is thus desirable that the mandatory jobs be determined dynamically to improve the energy-saving efficiency.

5.1 The general algorithm for dynamic approach

Our dynamic approach is shown in Algorithm 2. In general, our dynamic approach consists of two phases: an off-line phase followed by an online phase. During the offline phase, to ensure the (m,k)-constraints, the feasibility of the task set under R-pattern is tested according to Theorem 1 in [25]. Then based on it, for schedulable task sets, the speeds for the mandatory jobs for each task are scaled down as low as s_{crit} at the task level and will be used as predetermined speeds for the mandatory jobs. Then the blocking factor B_i for each task under the R-pattern is computed using the method similar to that in Theorem 1.

During the on-line phase, two job ready queues are maintained, *i.e.*, the mandatory job queue (MQ) and the optional job queue (OQ), with jobs in MQ always having higher priority than those in OQ. Upon arrival, a job, *i.e.*, $J_{ip} \in \tau_i$ is determined as mandatory job or optional job based on the execution results of the $k_i - 1$ jobs in the most recent history. It is determined as mandatory only if one more deadline miss will incur dynamic failure. The mandatory jobs in MQ will generally be executed with their speeds predetermined during the off-line phases. However, whenever there is zero or only one mandatory job in MQ, opportunities exist to update the execution speed for the upcoming mandatory jobs and save energy consumption more aggressively. More details of this method can be found in Section 5.2.

In our dynamic approach, not only the mandatory jobs but also the optional jobs can be executed. The execution of optional job has great potential in help reducing the overall energy consumption. For example, some optional job with actual execution time much shorter than its worst case can also meet its deadline with speed lower than its predetermined speed. And that will help to reduce the possibility of having to run mandatory jobs at higher processor speeds in the future. However, executing the optional job might incur extra energy cost, which needs to be addressed carefully. Otherwise, the energy reduction achieved from executing the optional job might not be

Algorithm 2 The algorithm for dynamic approach. (Algorithm *LKDN*)

```
1:  Upon job completion:
2:  if MQ is empty then
3:      $t_{cur}$ = the current time;
4:      $t_d$ = the latest starting time for the upcoming mandatory jobs according to Theorem 3 and Theorem 4 for
        R-pattern;
5:      if OQ is not empty then
6:          Select and run $J_i \in OQ$ with energy efficient speed $s'_i$ determined in Section 5.2 non-preemptively;
7:      else if $(t_d - t_{cur}) > T_{th}$ then
8:          Shut down the processor and set up the wake up timer to be $(t_d - t_{cur})$;
9:      end if
10: end if
11:
12: Upon job arrival or expiration of timer:
13: if MQ is not empty then
14:     if $J_i$ is the only job in MQ then
15:         Run $J_i$ with energy efficient speed $s'_i$ determined in Section 5.2 non-preemptively;
16:     else
17:         Run jobs in MQ with their current scaled speeds according to preemptive EDF;
18:     end if
19: end if
```

able to compensate the energy cost. Therefore, it is not only important to choose an appropriate optional job to execute, but also to determine the appropriate speed to execute the job. We introduce our method in choosing the optional job and its speed in section 5.3.

If no optional job is qualified to execute and the predicted idle interval is longer than the shut down threshold T_{th}, we shut down the processor and set the timer to be the idle interval length.

5.2 Update the predetermined speeds for mandatory jobs

When there exists only one mandatory job, i.e. J_i, in MQ, from Theorem 3 and Theorem 4, we can see that the speed of J_i can be scaled safely so long as J_i finishes no later than t_d. One comman strategy is to scale the speed of J_i as low as s_{crit}. If there is still available time, then the upcoming mandatory jobs are procrastinated and the processor is shut down if the length of idle time before t_d is larger than T_{th}. However, this strategy is not necessarily always the best strategy in saving energy.

Consider a task set consisting of two tasks ($\tau_1 = (20, 20, 5, 2, 4)$; $\tau_2 = (40, 40, 15, 1, 2)$). Assume the task set will be executed on the Intel XScale processor model [17]. According to [7], the power consumption function for Intel XScale [17] can be modeled approximately as $P_{act}(s) = 1.52s^3 + 0.08$ *Watt* by treating 1GHz as the reference speed 1. And the normalized critical speed in such a model is about 0.3 (at 297 MHz) with power consumption 0.12 *Watt* [7]. We assume the shut down overhead to be $E_o = 0.8mJ$ [7]. If the minimal processor speed is 0, the idle power consumption of the processor is 0.08 Watt and the corresponding shut down threshold is $T_{th} = 10ms$.

The scaled speed for the task set under E-pattern is 0.5 and the schedule of the static approach is shown in Figure 3(a). The energy consumption under the static approach is $(1.52 \times 0.5^3 + 0.08) \times 40 + (1.52 \times 0.3^3 + 0.08) \times 16.67 + 0.8 = 13.61mJ$. The scaled speed for the task set under R-pattern is 0.625. As shown in Figure 3(b), according to our optional job selection strategy in Section 5.3 (for brevity here we set the job selection control coefficient κ to be 1), our dynamic approach will choose to schedule the optional job for τ_2 first in the interval [0,40] and scale its speed to be 0.375 (after completion, its dynamic pattern is updated from 0 to be 1). Then at time $t = 40$, the dynamic approach will schedule the job J_{13} for τ_1. Since J_{13} is the only mandatory job in the ready queue and the latest starting time t_d for the future mandatory job(s) is 72, there is enough space to scale the speed for J_{13} to s_{crit} and shut down the processor upon completion of J_{13} at $t = 56.7$. Then at time $t = 72$, the processor will be waken up and continue to execute J_{14} at its predetermined speed 0.625. The energy consumption within the hyper period for the schedule in Figure 3(b) will be $(1.52 \times 0.375^3 + 0.08) \times 40 + (1.52 \times 0.3^3 + 0.08) \times 16.67 + 0.8 + (1.52 \times 0.645^3 + 0.08) \times 8 = 12.81mJ$.

However, a different schedule in Figure 3(c) which, instead of shutting down, uses the available space to help scale the speed of J_{14} to $s_{crit} = 0.3$ and keeps the processor idle for the rest of the time, has energy consumption of $(1.52\times0.375^3+0.08)\times40+0.12\times33.33+0.08\times6.67 = 10.92mJ$, which is 15.6% lower than that in Figure 3(b). Moreover, another schedule in Figure 3(d) uses the idle time to further reduce the speeds for job J_{13} and J_{14} to 0.25, which is below the critical speed, can achieve an additional 3.3% energy reduction compared with the schedule in Figure 3(c) (the energy consumption in Figure 3(d) is $(1.52 \times 0.375^3 + 0.08) \times 40 + (1.52 \times 0.25^3 + 0.08) \times 40 = 10.56mJ$).

In this example, we can see that the approach in Figure 3(b) considers only reducing the speed for the current job J_i. When looking forward, it might be more beneficial for the future mandatory jobs to compete for the available idle time to scale their speeds instead of shutting down the processor. With this in mind, we proposed a look-ahead approach which incorporates the speed information of the future mandatory jobs in scaling the speed of the current job J_i and shutting down the processor when necessary.

Specifically, when the current job J_i is ready, we temporarily scale its speed as low as s_{crit} first, *i.e*, set its current speed $s'_i = max\{\frac{c_i s_i}{(\min\{t_d, d_i\} - t_{cur})}, s_{crit}\}$. Then by considering the speed information for the future mandatory jobs arriving before t_d, this temporary speed s'_i might need to be updated depending on the expected finishing time $f_i(= t_{cur} + \frac{c_i s_i}{s'_i})$ of J_i under speed s'_i. (For easy of presentation, we call the future mandatory jobs arriving before t_d $look-ahead$ job set and denote it as $\mathcal{J}_{\mathcal{LA}}$. Although for each of the mandatory jobs in $\mathcal{J}_{\mathcal{LA}}$ we can consider them separately and eventually choose the one that can provide the best energy efficiency in the interval $[t_{cur}, t_d]$, to reduce the online time complexity, we only choose the mandatory job with the highest predetermined speed in $\mathcal{J}_{\mathcal{LA}}$ to be considered. For simplicity, we call it $look-ahead$ job and denote it as J_{la}.) Specifically, we need to consider three cases:

- 1) $(t_d - f_i) \le T_{th}$. In this case, it is not energy beneficial to shut down the processor at all. Instead, we should use the idle time to scale the speed of J_i and J_{la} as low as possible. Particularly, if $(t_d - f_i) < 0$ there is no need to update s'_i and the speeds for the jobs in $\mathcal{J}_{\mathcal{LA}}$ because the processor can continue to execute the new upcoming jobs upon completion of J_i and no new idle interval will be generated before t_d;

- 2) $(t_d - f_i) > T_{th}$ and $s_{la} = s_{crit}$. In this case, J_{la}'s speed is not higher than s_{crit} and the new idle interval generated between $[f_i, t_d]$ can be shut down if we procrastinate the upcoming mandatory jobs to t_d. So keeping the current value of s'_i and shutting down the processor while delaying the future mandatory jobs to t_d will be a good choice.

- 3) $(t_d - f_i) > T_{th}$ and $s_{la} > s_{crit}$. In this case, the idle interval generated between $[f_i, t_d]$ can be shut down if we procrastinate the upcoming mandatory jobs to t_d. However, we need to compare the beneficial between shutting down the processor and using the available time space to scale the speeds of J_i and J_{la}. In order to do so, we check whether both the speeds of J_i and J_{la} can be scaled to s_{crit} while the remaining idle time space within $[t_{cur}, t_d]$ is still long enough to shut down the processor. (This can be done by assuming the worst case execution time of J_{la} to be C_{la} and setting $s'_{la} = max\{\frac{C_{la} s_{la}}{(t_d - \max\{r_{la}, f_i\}) + C_{la})}, s_{crit}\}$). If the remaining time space $t_{rem} = (t_d - f_i - (C'_{la} - C_{la}))$ is longer than T_{th}, then we shut down the processor at f_i and set the wake up timer to be t_{rem} (here C'_{la} is the worst case execution time of J_{la} under s'_{la}). Otherwise there is not enough space to shut down the processor if J_{la} is scaled and we should reduce the speed of J_i and J_{la} as low as possible within $[t_{cur}, t_d]$, as we did in Figure 3(d).

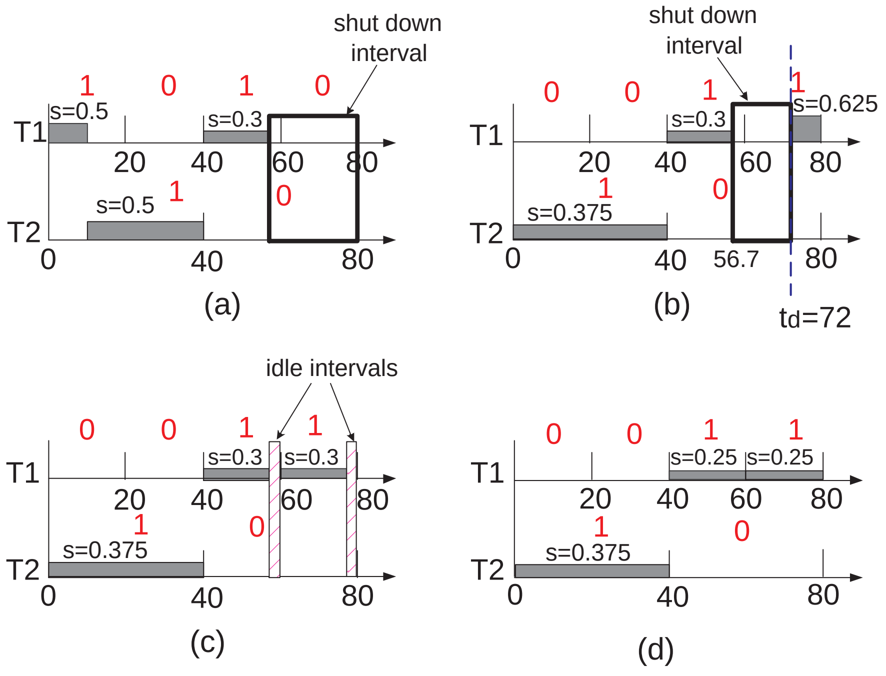


Figure 3. (a) The schedule of task set (τ_1=(20,20,5,2,4); τ_2=(40,40,15,1,2)) under the static approach; (b) The schedule under dynamic pattern with the *Scale-to-Critical-Speed-and-Shut-Down* Strategy; (c) The schedule under dynamic pattern with look-ahead approach; (d) The schedule under dynamic pattern and with speed reduced below the critical speed.

Note that in case 1) and 3), the speeds of J_i and J_{la} can be scaled below the critical speed s_{crit} (*e.g.*, job J_{13} and job J_{14} in Figure 3(d)). However, the energy efficiency can be better than the ones without looking-ahead. The overhead of this approach mainly comes from computing t_d which has been shown to have very low online overhead in [26].

5.3 Executing optional jobs

As stated before, when the mandatory queue MQ is empty, it can be energy beneficial to execute certain optional jobs in OQ. To choose the appropriate optional job to execute, we first introduce the following two definitions.

Definition 2 *The* **Task Energy Intensity** *of each task τ_i (denoted as TEI_i) is defined as*

$$TEI_i = \frac{m_i E(s_i)}{k_i T_i} \tag{8}$$

where $E(s_i)$ is the energy consumption to execute a mandatory job of τ_i under its predetermined speed s_i.

Definition 3 *The* **Job Energy Intensity** *of a job J_i (denoted as JEI_i) is defined as*

$$JEI_i = \frac{E(s_i')}{T_i} \tag{9}$$

where $E(s_i')$ is the energy consumption to execute job J_i under its current energy efficient speed s_i'.

Note that if the optional job cannot meet its deadline, it does not help in energy reduction or pattern adjustment. To guarantee the optional job can meet its deadline, we run the selected optional job non-preemptively. The potential energy efficient speed s_i' for each optional job J_i in OQ can be computed individually at the online phase by treating it as if it were the only ready job in OQ, similar to the speed determination approach in Section 5.2 (in this case the speed for the future mandatory jobs should also be updated correspondingly in the same way.)

Whenever a job completes and the MQ is empty, we first check each optional job J_i in OQ by inspecting its job energy intensity JEI_i under its potential energy efficient speed s_i' within the interval $[t_{cur}, t_d]$. Only those optional jobs with $JEI_i < \kappa TEI_i$ will be chosen as candidate jobs, where $\kappa(> 0)$ is a user defined parameter to fine tune the optional job selections. Generally κ can be set to be 1 for energy saving purpose. But the user can also vary the value of κ to control the chances that optional jobs will be scheduled and thus provide the micro-tunable performance (QoS *versus* energy) levels that can be achieved without affecting the schedulability of the whole task set.

For each candidate job, we can then calculate its energy-gain/criticality ratio as $\Delta E(J_i)/Cr_i$, where $\Delta E(J_i) = E(s_i) - E(s_i')$ and the criticality Cr_i is the number of deadline misses allowed before a dynamic failure occurs to J_i. Since the energy-gain $\Delta E(J_i)$ reflects the potential energy saving that can be achieved by executing the optional job and the criticality Cr_i reflects the relative urgency to schedule the optional job in order to meet the (m,k)-constraint, the candidate job with the maximal $\Delta E(J_i)/Cr_i$ will be chosen to be executed.

Compared with the static approach, the dynamic approach can lead to more aggressive energy reduction due to its adaptive pattern adjustment and speed determination with the run-time conditions. To ensure the (m,k)-constraints can be guaranteed, we have the following theorem (the proof is provided in the Appendix part E):

Theorem 5 *Let $\mathcal{T} = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$ be executed in a variable voltage processor , where $\tau_i = \{T_i, D_i, C_i, m_i, k_i\}$. The dynamic approach, with complexity of $O(n)$, can ensure the (m,k)-requirements for $\mathcal{T}$ if $\mathcal{T}$ is schedulable with R-pattern.*

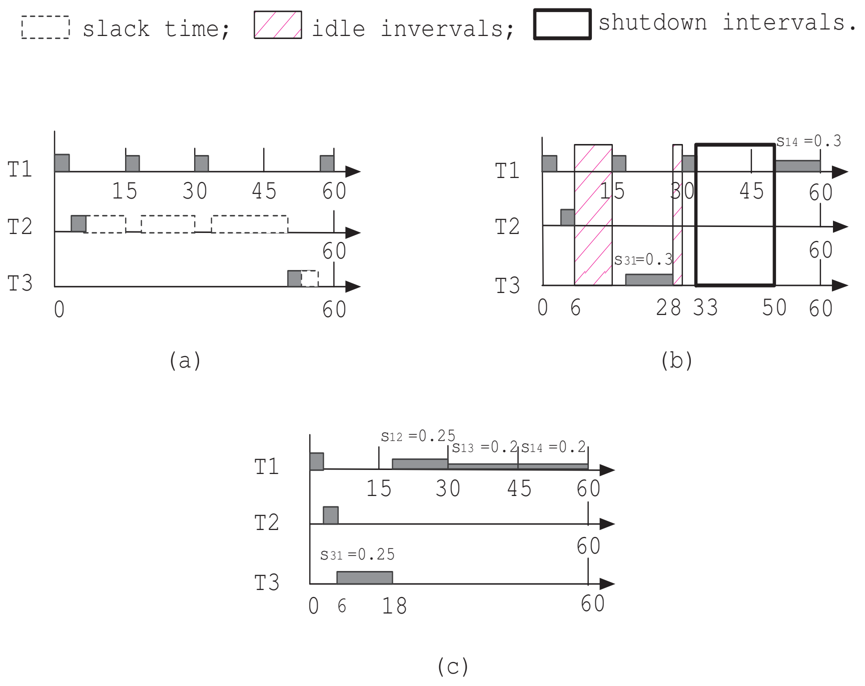


Figure 4. (a) The task set ($\tau_1 = (15, 15, 3, 1, 1)$; $\tau_2 = (60, 60, 42, 1, 1)$; $\tau_3 = (60, 60, 6, 1, 1)$) schedule with actual execution times ($c_{21} = 3$ and $c_{31} = 3$) and corresponding slack times; (b) The dynamic reclaiming schedule based on the algorithm in [21]; (c) The dynamic reclaiming schedule with our approach in Section 6.

6. Dynamic reclaiming

When there are more than one mandatory jobs in the ready queue and some of them present actual execution times shorter than their worst case, slack times can also be available to be reclaimed among mandatory jobs. In [21], the leakage-aware dynamic reclaiming approach was proposed to reduce the energy. The main idea is: assuming at time t, the processor is in sleeping state and the mandatory jobs J_i arrives with an amount of available slack $S_i(t)$ with priorities higher than or equal to it, the slack time $S_i(t)$ is used to reduce the speed of J_i to be no less than s_{crit} such that the active energy of J_i can be minimized (if $S_i(t)$ is not used up, the residue part can be used for dynamic procrastination).

However, minimizing the active energy alone might not be sufficient to reduce the total energy consumption.

Consider a task set of three tasks $\{\tau_1 = (15,15,3,1,1); \tau_2 = (60,60,42,1,1); \tau_3 = (60,60,6,1,1)\}$ to be executed on the same Intel XScale processor model as above, the task worst case execution schedule, actual execution times and slack times are shown in Figure 4(a). According to [21], the procrastination time Z_i for each task are $Z_1 = Z_2 = Z_3 = 0$. Also since the total utilization of the task set is 1, the static speed for each task is 1.

As shown in Figure 4(b), at time $t = 6$, job J_{31} began to execute with slack time $S_3(t) = 39$ time units from higher priority job. Since the static speed s_{31} of job J_{31} is 1, it could be reduced to the critical speed s_{crit} safely by reclaiming 7 time units from the slack time available. Since there is still slack time left, the rest slack time can be used to procrastinate J_{31}. However, at $t = 15$, since $Z_1 = 0$ and the slack times from job J_{21} and J_{31} did not have priorities higher than or equal to job J_{12}, according to [21], J_{12} could not be procrastinated in this case. The same situation also happened to J_{13} at $t = 30$. Only when J_{14} arrived at $t = 45$, since the rest slack times from job J_{21} and J_{31} had higher priorities than J_{14}, they could be used to reduce the speed of J_{14} to the citical speed s_{crit}. And the residue part of the slack could be used to procrastinate J_{14} to $t = 50$. As a result, the idle interval generated, *i.e.*, $[33,50]$ could be shut down. The total energy consumption within the interval $[0, LCM(T_i)]$ is $E = (1.52\times 1^3 + 0.08)\times 6 + 0.08\times 9 + (1.52\times 1^3 + 0.08)\times 3 + (1.52\times 0.3^3 + 0.08)\times 10 + 0.08\times 2 + (1.52\times 1^3 + 0.08)\times 3 + 0.8 + (1.52\times 0.3^3 + 0.08)\times 10 = 21.84mJ$. However, a different schedule in Figure 4(c) which utilizes the slack time $S_3(6)$ available to J_{31} at $t = 6$ to reduce the speed of job J_{31} and the look-ahead job J_{12} generated total energy consumption within the interval $[0, LCM(T_i)]$ as $E = (1.52\times 1^3 + 0.08)\times 6 + (1.52\times 0.25^3 + 0.08)\times 24 + (1.52\times 0.2^3 + 0.08)\times 15\times 2 = 14.97mJ$, which is 31.4% lower than that from Figure 4(b). Note that in the schedule of Figure 4(c), at $t = 30$, even the slack time from the early completion of J_{21} does not have priority higher than J_{13}, but it can still be reclaimed by our algorithm to reduce the speed for the current job J_{13} and the look-ahead job J_{14}.

In this example, we can see that when reclaiming the slack time, it might be more beneficial to look into the future jobs to achieve better energy saving performance. With this in mind, we propose a look-ahead dynamic reclaiming approach by incorporating the speed information of the future jobs. Specifically, when the current job J_i is ready with reclaimable slack time $S_i(t)$, we also temporarily scale its speed as low as s_{crit} first, *i.e*, set its current speed s_i to be $\tilde{s}_i = \max\{\frac{c_i\times s_i}{c_i+S_i(t)}, s_{crit}\}$. Then the expected finishing time (denoted as $F_{\mathcal{J}}$) for all uncompleted mandatory jobs in the ready queue (denoted as $\mathcal{J}$, and $J_i \in \mathcal{J}$) can be computed as $F_{\mathcal{J}} = t_{cur} + \sum_{J_p\in\mathcal{J}} c_p$, where c_p is the remaining actual execution time of job J_p under its current scaled speed. After that, depending on the value of $F_{\mathcal{J}}$, the current speeds for J_i and other mandatory jobs in $\mathcal{J}$ as well as that for the look ahead job J_{la} might need to be updated, which can be classified into three cases:

- 1) $(t_d - F_{\mathcal{J}}) \le T_{th}$. In this case, it is not energy beneficial to shut down the processor at all and we should use the slack/idle time to scale the speeds of jobs in $F_{\mathcal{J}}$ and J_{la} as low as possible. The speeds of the jobs in $\mathcal{J}$ can be reduced in the same way as case 1) in Section 5.2.

- 2) $(t_d - F_{\mathcal{J}}) > T_{th}$ and $s_{la} = s_{crit}$. In this case, J_{la}'s speed is not higher than s_{crit} and the new idle interval generated between $[F_{\mathcal{J}}, t_d]$ can be shut down if we procrastinate the upcoming mandatory jobs to t_d. So keeping the current value of s'_i and shutting down the processor while delaying the future mandatory jobs to t_d will be a good choice.

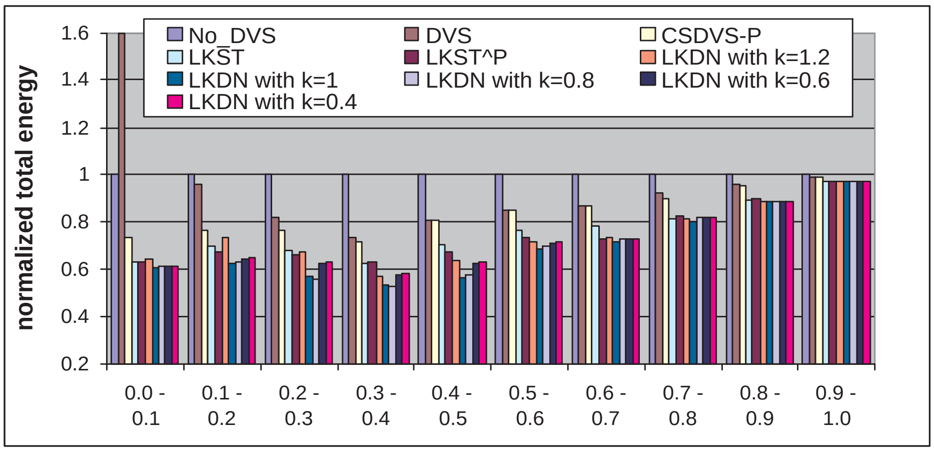
- 3) $(t_d - F_{\mathcal{J}}) > T_{th}$ and $s_{la} > s_{crit}$. In this case, the idle interval generated between $[F_{\mathcal{J}}, t_d]$ can be shut down if we procrastinate the upcoming jobs to t_d. However, we need to compare the beneficial between shutting down the processor and using the available slack time to scale the speeds of J_{la}. In order to do so, we need to check whether both the speeds of J_i and J_{la} can be scaled to s_{crit} while the remaining idle time space within $[t_{cur}, t_d]$ is still long enough to shut down the processor. This can be done by setting $s'_{la} = max\{\frac{C_{la}s_{la}}{(t_d - \max\{t_a, F_{\mathcal{J}}\}) + C_{la})}, s_{crit}\}$ first. If the remaining time space $t_{rem} = (t_d - F_{\mathcal{J}} - (C'_{la} - C_{la}))$ is longer than T_{th}, then we shut down the processor at $F_{\mathcal{J}}$ and set the wake up timer to be t_{rem} (here C'_{la} is the worst case execution time of J_{la} under s'_{la}). Otherwise there is no enough space to shut down the processor if J_{la} is scaled and we should reduce the speeds of jobs in $F_{\mathcal{J}}$ and J_{la} as low as possible within $[t_{cur}, t_d]$, as in case 1).

Note that in case 1) and 3), it is also possible that the speeds of jobs in $F_{\mathcal{J}}$ and J_{la} can be reduced to below the critical speed s_{crit}, (*e.g.*, job J_{31} and job J_{12} in Figure 4(c)). Moreover, another difference between our dynamic approach and the approach in [21] is that our approach does not require the slack time $S_i(t)$ available to the current job J_i have higher priority than J_i. So the previous aggressive slack reclaiming approach that can steal slack from lower priority jobs (e.g., [3, 22]) can be incorporated into our approach to achieve better energy efficiency.

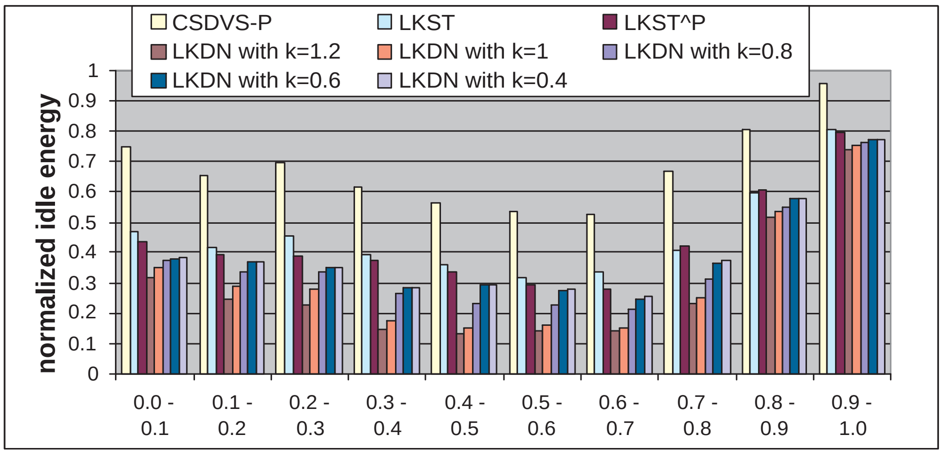
7 Experimental results

Six different approaches are studied using experiments in this section. In the first approach, the task sets are statically partitioned with E-pattern, and the mandatory jobs are executed with the highest processor speed. We refer this approach as *NoDVS* and use its results as the reference results. In the second approach, the task sets are statically partitioned with E-pattern, but the speeds of the tasks are scaled down at the task level as low as possible, We refer this approach as *DVS*. In the third approach, the task sets are statically partitioned with E-pattern, and the speed of the tasks are scaled as low as s_{crit} at the task level, then the approach in [21] based on individual task procrastination is applied, we call this approach *CSDVS-P*. The fourth approach, namely *LKST*, is our static approach introduced in Section 4. The task sets are statically partitioned with *E-pattern*, and the speeds of the tasks are scaled as low as s_{crit}. When the system is idle, we tried to delay the mandatory jobs with the approaches introduced in Section 4 and shut down the processor whenever necessary. Since the parameter controlled procrastination approach in [7] can also be applied to our static approach, in the fifth approach, we will also incorporate this strategy into our static approach with the parameter set correspondingly (according to [7], the best value should be around 0.6 for our processor model), we refer this approach as $LKST^P$. The sixth approach *LKDN* is our dynamic approach as explained in Section 5. In this approach, we dynamically vary the job patterns according to the run time conditions and reduce the job speed (below the critical speed when necessary) as well as shut down the processor whenever possible. Since the value of κ, *i.e.*, the job selection coefficient in our dynamic approach, can affect the selection of optional jobs and thus the performance of the algorithm, we also vary the value of κ within the range of (0.4, 0.6, 0.8, 1.0, 1.2) to compare their performance. For the processor model we will adopt the same processor model as used in [7], *i.e.* the Intel XScale processor. The parameters are the same as used in Section 5.2.

We first studied the energy consumption of these approaches based on the synthesized task sets. The periodic task sets tested in our experiments were randomly generated with the periods and the worst case execution times (WCET) of the tasks randomly chosen in the range of $[10ms, 100ms]$ and $[1ms, 30ms]$, respectively. The deadlines of the tasks were set to be less than or equal to their periods. The actual execution time of a job was randomly picked from [0.1WCET, WCET]. The m_i and k_i for the (m,k)-constraints were also randomly generated such that k_i is uniformly distributed between 2 to 10, and $m_i \leq k_i$. The total utilization, i.e., $\sum_i \frac{m_i C_i}{k_i T_i}$, is divided into intervals



(a)



(b)

Figure 5. (a) The total energy comparison by the different approaches. (b) The idle energy comparison by the different approaches.

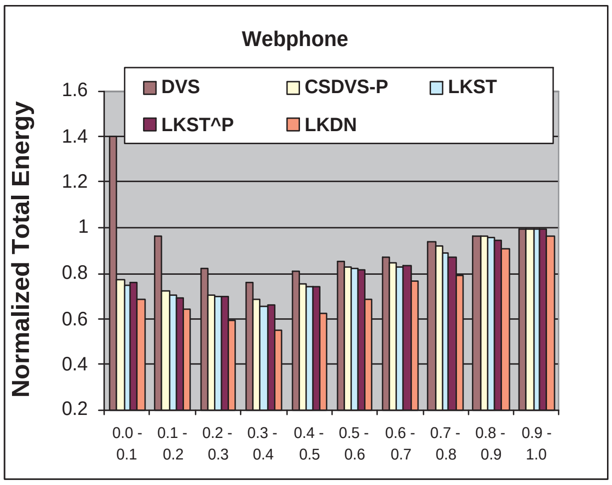
of length 0.1 and we randomly generate 50 feasible task sets for each interval. The energy consumption for each approach was normalized to that by *NoDVS*, and the results are shown in Figure 5(a) and (b).

From Figure 5(a), *DVS* will cause dramatic increase in the total energy consumption for low utilization intervals. For example, when the utilization is between [0.0, 0.1], the energy consumption by *DVS* is more than 60% of that by *NoDVS*. On the contrary, voltage scaling approaches with static energy and procrastination in mind can reduce the total energy consumption significantly. Moreover, with effective procrastination for the mandatory jobs and dynamic pattern adjustment, our static approach and dynamic approach can reduce the energy consumption further effectively. For example, compared with *CSDVS-P*, *LKST* can lead to a total energy reduction by around 10%. The $LKST^P$ has a marginal improvement over *LKST* and its energy consumption in some intervals can be slightly higher than that by *LKST*. Our dynamic approach *LKDN* can reduce the total energy more significantly. Compared with *CSDVS-P*, the maximal reduction can be around 26%. It is also noticed that during some low utilization interval the total energy consumption by *LKDN* is close to *LKST* and $LKST^P$ for small κ values. This is because, when the the system utilization is low and the value of κ is small, there is little chance to select candidate optional jobs and scale the speeds further. In this case, the procrastination and shut-down strategy will dominate. In some intervals, the energy performance of *LKDN* with very large value of κ is not the best either because in those cases *LKDN* optionally executed some redundant jobs. Although those jobs can help enhance the user perceivable QoS levels, they might also cost extra energy that cannot be completely compensated by the energy reduction from varying the job patterns. And it seems the overall energy is the lowest when κ is between [0.8, 1] and generally setting κ to be around 1 is reasonably good.

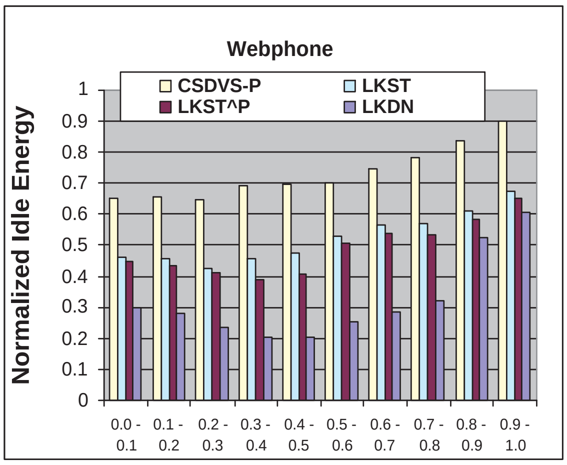
Due to the energy overhead of DPD and the dramatic increase of the static power, the energy consumption during the processor idle time will also account for a significant part of the total energy consumption. We are therefore interested in investigating how our approach can help reduce this part of energy. Figure 5(b) shows the average idle energy consumptions by the different approaches. Note that, our approaches, *i.e.*, *LKST*, $LKST^P$ and *LKDN* can always lead to much better idle energy savings than the previous approaches. *LKST* and $LKST^P$ both consume much lower idle energy than *CSDVS-P* due to effective idle extension and merging. The performance of *LKDN* is even better because it utilizes the idle intervals more efficiently to help reduce the job speed (below the critical speed when necessary) and to facilitate shut-down. Compared with *CSDVS-P*, *LKST* and $LKST^P$ can reduce the idle energy by up to around 40% and *LKDN* can reduce the idle energy by up to around 78%. Also note that when the value of κ is relatively larger, the idle energy consumption for *LKDN* is relatively lower because in this case there are more chances that the idle time will be utilized to schedule the optional jobs. However, setting the κ value as too high, for example, let $\kappa > 1$, might cause too many optional jobs to be scheduled and increase the total energy consumption. Considering that, setting the value of κ to be around 1.0 tends to provide better performance in minimizing the total energy and idle energy consumption simultaneously.

Next, we tested our techniques in a more practical environment. The test cases contained two real world applications: webphone [40], and INS (Inertial Navigation System) [1]. The timing parameters such as the deadlines, periods, and execution times were adopted from these practical applications. The actual execution times and the (m,k)-constraints were generated as we did for the synthesized task sets. Since the value of κ around 1.0 tends to have better performance for our dynamic approach, this time we fixed the value of κ to be 1.0. The normalized total energy consumptions and idle energy consumptions are shown in Figure 6.

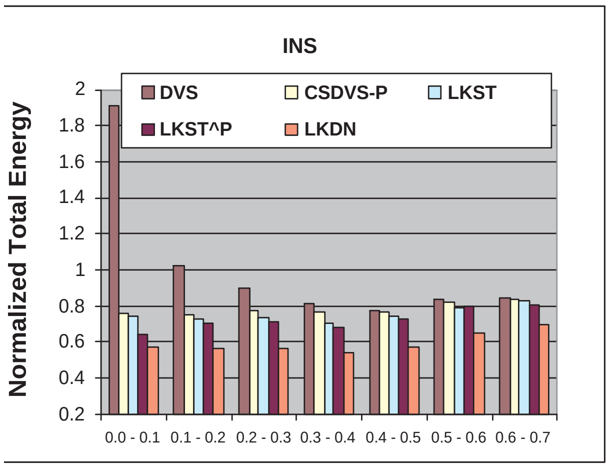
The experimental results based on the practical applications further demonstrate the effectiveness of our approaches in saving energy. As shown in Figure 6(a) and (b), for the webphone application, the static approach *LKST* and $LKST^P$ can reduce the total energy consumption by around 8% and idle energy consumption by about 30%. And the energy saving by the dynamic approach (*LKDN*) can be up to 15% for total energy consumption and up to 71% for idle energy consumption. For INS application, as shown in Figure 6(c) and (d), the static approach *LKST* and $LKST^P$ can reduce the total energy consumption by about 12% and idle energy consumption by about 36%. And the energy saving by *LKDN* can be up to 23% for total energy consumption and 80% for idle energy consumption.



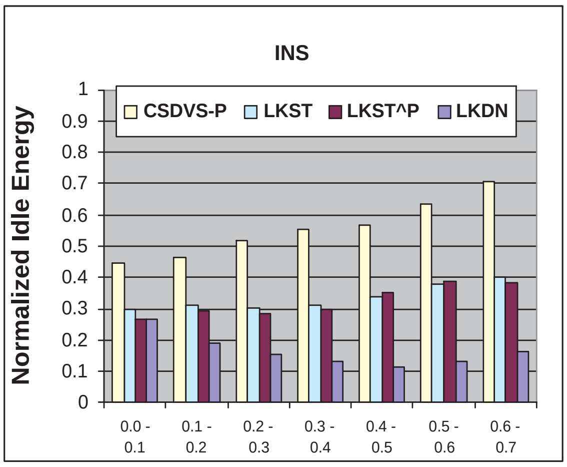
(a)



(b)



(c)



(d)

Figure 6. (a) The total energy comparison for webphone. (b) The idle energy comparison for webphone. (c) The total energy comparison for INS. (d) The idle energy comparison for INS.

8 Conclusions and future work

Low power/energy consumption and QoS guarantee are two of the most critical factors for the successful design of pervasive real-time computing platforms. While the dynamic voltage scaling (DVS) techniques are efficient in reducing the dynamic power consumption for the processor, varying voltage alone becomes less effective for the overall energy reduction as the static power is growing rapidly. In this paper, we propose two approaches, a static one and a dynamic one, to reduce the overall energy consumption for real-time systems while guaranteeing the given QoS requirement in terms of (m,k)-constraints. The static approach determines the mandatory jobs statically and try to delay the mandatory jobs and shut down the processor whenever possible. The dynamic approach vary the job pattern dynamically during the runtime and determine the job speed in a more adaptive way. Through extensive simulations, our approaches outperformed previous research in both overall and idle energy reduction while providing the (m,k)-guarantee.

For the future work, we plan to expand our current research in the following directions:

- As we know, although our dynamic approach is efficient in reducing the overall energy consumption, it is very hard to predict the feasibility of the mandatory jobs accurately under the dynamic pattern. In this paper, we have adopted R-pattern to provide a bottom line of guaranteeing the schedulability of the mandatory jobs in the worst case (as shown in the proof of Theorem 5). However, this strategy can be kind of conservative as in practice the worst case might never happen. So in part of our future work we are trying to find better ways to predict the feasibility of our dynamic approach more accurately and thus further improve its energy efficiency correspondingly.
- As shown in [43, 37], there is a dependency relationship between leakage and temperature and it plays an important role in both the power and thermal aware design. As part of our future work, we will explore minimizing the overall energy with leakage-temperature dependency awareness while still ensuring the (m,k)-guarantee.
- Since multi-core processors are becoming popular to improve the system performance for next generation real-time embedded systems, we would also tackle the issues of leakage-aware energy minimization on multi-core platforms with (m,k)-constraints.

References

[1] A.Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21:920–934, May 1995.

[2] T. A. AlEnawy and H. Aydin. Energy-constrained scheduling for weakly-hard real-time systems. *RTSS*, 2005.

[3] H. Aydin, R. Melhem, D. Mosse, and P. Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *ECRTS01*, June 2001.

[4] H. Aydin, R. Melhem, D. Mosse, and P. Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *RTSS01*, December 2001.

[5] G. Bernat and A. Burns. Combining (n,m)-hard deadlines and dual priority scheduling. In *RTSS*, Dec 1997.

[6] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power cmos digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[7] J.-J. Chen and T.-W. Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *ICCAD*, 2007.

[8] J.-J. Chen, N. Stoimenov, and L. Thiele. Feasibility analysis of on-line dvs algorithms for scheduling arbitrary event streams. In *RTSS*, 2009.

[9] J.-J. Chen and L. Thiele. Expected system energy consumption minimization in leakage-aware dvs systems. In *ISLPED*, 2008.

[10] J.-J. Chen and L. Thiele. Energy-efficient scheduling on homogeneous multiprocessor platforms. *SAC'10 (PADO Track)*, 2010.

[11] M.-I. Chen and K.-J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Journal of Real-Time Systems*, 2(4):325–346, 1990.

[12] V. Devadas and H. Aydin. On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications. *IEEE Transactions on Computers*, 61(1):31–44, 2012.

[13] L. George, D. D. Voluceau, and B. L. C. C. (france). Preemptive and non-preemptive real-time uni-processor scheduling, 1996.

[14] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computes*, 44:1443–1451, Dec 1995.

[15] S. Hua and G. Qu. Energy-efficient dual-voltage soft real-time system with (m,k)-firm deadline guarantee. In *CASE'04*, 2004.

[16] S. Hua, G. Qu, and S. Bhattacharyya. Energy reduction techniques for multimedia applications with tolerance to deadline misses. *DAC*, pages 131–136, 2003.

[17] INTEL-XSCALE. http://developer.intel.com/design/xscale/. 2003.

[18] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *ISLPED*, 1998.

[19] ITRS. *International Technology Roadmap for Semiconductors*. International SEMATECH, Austin, TX., http://public.itrs.net/.

[20] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. *DAC*, 2004.

[21] R. Jejurikar, C. Pereira, and R. Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. *DAC*, 2005.

[22] W. Kim, J. Kim, and S.L.Min. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack analysis. *DATE*, 2002.

[23] F. Kong, Y. Wang, Q. Deng, and W. Yi. Minimizing multi-resource energy for real-time systems with discrete operation modes. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, ECRTS '10, pages 113–122, Washington, DC, USA, 2010.

[24] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *RTSS*, pages 201–209, 1990.

[25] N. Linwei. Energy-aware dual-mode voltage scaling for weakly hard real-time systems. *SAC'10 (Real Time System Track)*, 2010.

[26] N. Linwei and G. Quan. Reducing both dynamic and leakage energy consumption for hard real-time systems. *CASES'04*, Sep 2004.

[27] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 17(2):46–61, 1973.

[28] J. Liu. *Real-Time Systems*. Prentice Hall, NJ, 2000.

[29] S. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microporcessor under dynamic workloads. *ICCAD*, 2002.

[30] B. Mochocki, X. Hu, and G. Quan. A realistic variable voltage scheduling model for real-time applications. *ICCAD*, 2002.

[31] L. Niu and G. Quan. Energy minimization for real-time systems with (m,k)-guarantee. *IEEE Trans. on VLSI, Special Section on Hardware/Software Codesign and System Synthesis*, pages 717–729, July 2006.

[32] L. Niu and G. Quan. Peripheral-conscious scheduling on energy minimization for weakly hard real-time systems. *DATE*, 2007.

[33] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP*, 2001.

[34] X. Qi, D. Zhu, and H. Aydin. Global scheduling based reliability-aware power management for multiprocessor real-time systems. *Real-Time Syst.*, 47:109–142, March 2011.

[35] G. Quan and X. Hu. Enhanced fixed-priority scheduling with (m,k)-firm guarantee. In *RTSS*, pages 79–88, 2000.

[36] G. Quan, N. Linwei, X. S. Hu, and B. Mochocki. Real time scheduling for reducing overall energy on variable voltage processors. *International Journal of Embedded System: Special Issue on Low Power Embedded Computing*, 4(2):127–140, 2009.

[37] G. Quan and Y. Zhang. Leakage aware feasibility analysis for temperature-constrained hard real-time periodic tasks. In *ECRTS*, 2009.

[38] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for qos management. In *RTSS*, Dec. 1997.

[39] P. Ramanathan. Overload management in real-time control applications using (m,k)-firm guarantee. *IEEE Trans. on Paral. and Dist. Sys.*, 10(6):549–559, Jun 1999.

[40] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers*, 18(2), March-April 2001.

[41] T.P.Baker. Stack-based scheduling for real-time processes. *Real-Time Systems*, 3:67–99, 1991.

[42] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *AFCS*, pages 374–382, 1995.

[43] L. Yuan and G. Qu. Alt-dvs: Dynamic voltage scaling with awareness of leakage and temperature for real-time systems. In *AHS*, 2007.

[44] B. Zhao and H. Aydin. Minimizing expected energy consumption through optimal integration of dvs and dpm. In *ICCAD*, 2009.

[45] B. Zhao, H. Aydin, and D. Zhu. On maximizing reliability of real-time embedded applications under hard energy constraint. *IEEE Trans. Industrial Informatics*, pages 316–328, 2010.

[46] D. Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. *ACM Trans. Embed. Comput. Syst.*, 10:26:1–26:27, January 2011.

[47] Y. Zhu and F. Mueller. Dvsleak: combining leakage reduction and voltage scaling in feedback edf scheduling. In *LCTES*, 2007.

A Proof for Theorem 1

Proof: Suppose at time t', the first deadline missing happened to a mandatory job from the E-pattern. Let t_0 be the last time before t' such that there are no pending mandatory jobs with release times before t_0 and deadlines before or at t'. (Since no job is released before time 0, t_0 is well defined.) Let Φ be the set of periodic tasks that have periodic instances with release times and deadlines in $[t_0, t']$. Then by choice of t_0 and t' we have $\Phi \subseteq \{\tau_1, ..., \tau_{l(t'-t_0)}\}$. Then according to Baker's argument in [41], the only mandatory jobs that can execute in $[t_0, t']$ are those of tasks in Φ, plus at most one instance of a task τ_j $(j > l(t'-t_0))$ that may block a task in Φ. Moreover, the total length of time that τ_j can execute in $[t_0, t']$ is bounded by the longest time it uses a resource, which, by definition, is bounded by B_i for each task τ_i in Φ and also bounded by $B_{l(t'-t_0)}$ in particular.

In $[t_0, t']$, there is no idle time. And for every task τ_i in Φ, the work-demand in $[t_0, t']$ is bounded by $(\lceil \frac{m_i}{k_i} \lfloor \frac{t'-t_0+T_i-D_i}{T_i} \rfloor \rceil) C_i$. Since there is an overflow, the total work-demand for processor time in $[t_0, t']$ exceeds $(t'-t_0)$. So, we have

$$\sum_{D_i \leq (t'-t_0)} (\lceil \frac{m_i}{k_i} \lfloor \frac{t'-t_0+T_i-D_i}{T_i} \rfloor \rceil) C_i + B_{l(t'-t_0)} > (t'-t_0) \tag{10}$$

let $t = t' - t_0$, we have

$$\sum_{D_i \leq t} (\lceil \frac{m_i}{k_i} \lfloor \frac{t+T_i-D_i}{T_i} \rfloor \rceil) C_i + B_{l(t)} > t \tag{11}$$

which contradicts Equation (3). □

B Proof for Theorem 2

Proof: If we treat the processor itself as a shared resource that each of the task in $\mathcal{T}$ can access, then the concurrency control protocols which B_i is based on can be reduced to our task procrastination algorithm. The idea is: suppose we construct a virtual task $\tilde{\tau}$ with a non-preemptable critical section $\tilde{cs}$ and with an infinite deadline. If we add $\tilde{\tau}$ to the original task set $\mathcal{T}$, obviously $\tilde{\tau}$ will always stay in the lowest priority level. Whenever the system is idle at time t, the virtual task $\tilde{\tau}$ will be started and enter its critical section $\tilde{cs}$, which, by definition of the blocking factor B_i, is bounded by the value of B_i computed by (4) for each task τ_i in $\mathcal{T}$.

Given the assumption above, for a mandatory J_i of task τ_i arriving at r_i, the procrastination of J_i can be transferred to the operation of the concurrency control protocols as followed: if the system is idle at r_i^-, it is experiencing blocking from $\tilde{\tau}$ started at r_i^-. Otherwise, it is experiencing blocking from another lower priority mandatory job executing at r_i^-. First, we show that such kind of procrastination on J_i can guarantee the deadlines of all mandatory jobs. We use contradiction.

Suppose at time t', the first deadline missing happened to a mandatory job from the E-pattern. Let t_0 be the last time before t' such that there are no pending mandatory jobs with release times before t_0 and deadlines before or at t'. (Since no job is released before time 0, t_0 is well defined.) Let Φ be the set of periodic tasks that have periodic instances with release times and deadlines in $[t_0, t']$. Then by choice of t_0 and t' we have $\Phi \subseteq \{\tau_1, ..., \tau_{l(t'-t_0)}\}$. Then according to Baker's argument in [41], the only mandatory jobs that can execute in $[t_0, t']$ are those of tasks in Φ, plus at most one instance of a task τ_j that may block a task in Φ. There are two cases:

- 1) The system is idle at t_0^-: in this case, since t_0 must be the arrival time for some task τ_x in Φ, τ_j will be the virtual task $\tilde{\tau}$ which will block/procrastinate some task in Φ for at most $\tilde{cs}$ time units, which, by definition, is bounded by $B_l(t'-t_0)$.
- 2) The system is busy at t_0^-: in this case, τ_j will be some task in $\mathcal{T}$ with deadline $D_j > (t'-t_0)$. And according to to Baker's argument in [41], the maximal time τ_j can execute during the interval $[t_0, t']$ is bounded by B_i for each task τ_i in Φ and also by $B_l(t'-t_0)$ in particular.

In $[t_0, t']$, there is no idle time after we introduce the virtual task $\tilde{\tau}$. And for every task τ_i in Φ, the work-demand in $[t_0, t']$ is bounded by $(\lceil \frac{m_i}{k_i} \lfloor \frac{t'-t_0+T_i-D_i}{T_i} \rfloor \rceil) C_i$. Since there is an overflow, the total work-demand for processor time in $[t_0, t']$ exceeds $(t'-t_0)$. So, we have

$$\sum_{D_i \le (t'-t_0)} (\lceil \frac{m_i}{k_i} \lfloor \frac{t'-t_0+T_i-D_i}{T_i} \rfloor \rceil) C_i + B_{l(t'-t_0)} > (t'-t_0) \tag{12}$$

let $t = t'-t_0$, we have

$$\sum_{D_i \le t} (\lceil \frac{m_i}{k_i} \lfloor \frac{t+T_i-D_i}{T_i} \rfloor \rceil) C_i + B_{l(t)} > t \tag{13}$$

which contradicts Equation (3).

Next we show J_i can be further procrastinated up to B_i time units after its arrival time if necessary.

Note that, in the above argument, due to the property of the concurrency control protocol, each mandatory job J_i can only be blocked by $\tilde{\tau}$ or other lower priority job by at most once. So the actual procrastination time X_i of J_i can be less than B_i. However, in this case we can always procrastinate J_i further by switching execution time between J_i and other job(s) executing between $[r_i + X_i, d_i]$. It is not to see that in EDF schedule such kind of execution time switching is safe for both J_i and the other job(s) so long as the total amount of switching doesn't exceed $(B_i - X_i)$ time units. If the amount of switching equals $(B_i - X_i)$, J_i is eventually procrastinated by a total of B_i time units and all job deadlines are still guaranteed. □

C Proof for Theorem 3

Proof: From Equation (15), we have

$$T_{LS}(\mathcal{J}') \le (r_i + B_i), i = 0, 1, ..., n-1, \tag{14}$$

Thus we have

$$T_{LS}(\mathcal{J}') - r_i \le B_i, i = 0, 1, ..., n-1, \tag{15}$$

which means that, after delay, none of the mandatory job J_i in $\mathcal{J}'$ can be procrastinated more than B_i time units. By Theorem 2, no mandatory job in $\mathcal{M}$ will miss its deadline. □

D Proof for Theorem 4

Proof: To prove Theorem 1, we first introduce the following lemma.

Lemma 1 *[30] Let $\mathcal{M}$ be the mandatory job set based on E-pattern. Let*

$$lst(J_i) = d_i - \sum_{J_k \in hp(J_i)} \frac{c_k}{s_k}, \tag{16}$$

where $hp(J_i)$ is the jobs with the same or higher priorities than that of J_i. Then, the latest starting time that jobs in $\mathcal{M}$ can be delayed to without causing any deadline missing is

$$LST(\mathcal{J}) = \min_i \{lst(J_i)\}. \tag{17}$$

The rationale behind Lemma 1 is that if the accumulated workload from a mandatory job and *all* the higher priority mandatory jobs can be finished between the starting time and its deadline, the deadline of this job can be satisfied. In addition, the minimal latest starting time of all the mandatory jobs can certainly guarantee all the deadlines.

In [26], Niu et al. extended Lemma 1 in computing the latest starting time based on information from only a subset $\mathcal{J}_s$ of the jobs in $\mathcal{M}$ that arrive before the earliest deadline of upcoming mandatory jobs (so called delay bound and denoted as T_B). This approach has a much lower complexity and hence is more suitable for on-line purpose. However, as pointed out in [26], the latest starting time computed by employing Equation (16) only for jobs arriving before T_B may not be valid since the validity of latest starting time in Lemma 2 is ensured by employing (16) for every mandatory job in $\mathcal{M}$. In this regard, Niu et al. proposed to use the *effective deadline* of a job (i.e. the time before which a job has to be finished such that it will not cause any other job to miss deadline) in place of the deadline in (16). To keep low complexity of the algorithm, they simply defined the effective deadline for a job by its own deadline or the earliest arrival time of the upcoming low priority mandatory job, whichever is smaller.

Here in Theorem 4, in order to get larger value of latest starting time, we prolong the *effective deadline* for each mandatory job in $\mathcal{J}_s$ with the blocking factor B_p of the next upcoming mandatory jobs. And the proof of it can also be done based on Theorem 2, similar to that of Theorem 3. Specifically, assume the execution of all the upcoming mandatory jobs in $\mathcal{J}'$ is delayed to $T_{LS}(\mathcal{J}')$, for each unfinished mandatory job J_i in $\mathcal{M}$, we consider two case:

- 1) $r_i > T_B$: in this case, by the definition of effective deadline in Equation (7), similar to the argument in Theorem 3, J_i will be procrastinated by no more than B_i time units. By Theorem 2, the schedulability of J_i can be guaranteed.
- 2) $r_i \leq T_B$: in this case, from Lemma 1, it is easy to see that the schedulability of J_i can also be guaranteed.

□

E Proof for Theorem 5

Proof: The worst case scenario of Algorithm 2 happens when at certain time point t, each task τ_i already had $k_i - m_i$ missed deadlines before t. Then the next m_i jobs of each task τ_i should be designated as mandatory jobs consecutively in order to meet the (m,k)-constraint, which is equivalent to the R-pattern. Since $\mathcal{T}$ is schedulable with R-pattern under their predetermined speed, that means even under the worst case, the deadlines of all mandatory jobs of Algorithm 2 can still be guaranteed.

the complexity of our dynamic approach mainly comes from three cases:

- 1) the current mandatory job queue (MQ) is empty: in this case we need to select the appropriate optional job from the optional job queue (OQ) to execute;
- 2) the current mandatory queue MQ is not empty but there is only one mandatory job in it: in this case we might need to update the speed of the current mandatory job;
- 3) there is more than one mandatory jobs in the current mandatory queue MQ: in this case, we just follow the general preemptive EDF scheme in scheduling the mandatory jobs in MQ with their current speeds.

In case 1) and case 2), we need to compute the latest starting time t_d for the upcoming mandatory jobs as well as updating the speed of the look-ahead job. Give B_i for task τ_i can be computed offline, the online overhead of computing t_d based on to Theorem 3 takes at most $O(n)$ time. Moreover, as shown in [26], the online overhead of computing t_d based on to Theorem 4 is usually very small for periodic task sets. So the main overhead comes from the number of context switches in choosing the optional job to execute (in case 1)) or updating the speed of the current mandatory job (in case 2)) as well as updating the speed for the look-ahead job if necessary. Since in our algorithm each time we only need to consider at most n optional jobs in OQ or 1 mandatory job in MQ and at most 1 look-ahead job, the online complexity will not exceed $O(n)$.

In case 3), the online complexity will not exceed $O(n)$, either. □