System Wide Dynamic Power Management for Weakly Hard Real-Time Systems

Abstract

Energy reduction is critical to increase the mobility and battery life for today's pervasive computing systems. At the same time, energy reduction must be subject to the real-time constraints and quality of service (QoS) requirements for applications running on these systems. This paper presents a novel run-time approach to reduce the system-wide *energy consumption for such systems using dynamic power management. In this paper, the applications are modeled using a popular weakly hard real-time model, i.e., the* (m,k)*-model, which requires that at least m out of any k consecutive instances of a task meet their deadlines. Our experimental results show that, by judiciously scheduling the real-time tasks and shutting down the processor and/or peripheral devices, our approach can lead to significant energy savings while guaranteeing the* $(m,k)-$*firm deadlines at the same time.*

Keywords: Dynamic power management, weakly hard real-time, power-aware scheduling, Quality of Service, I/O devices

1 Introduction

Energy conservation has come to be recognized as a critical issue in design of pervasive real-time embedded systems, particularly due to the proliferation of mobile systems with limited power resources. For the sake of mobility, these portable computing and communication devices require low energy consumption to maximize the battery lifetime. As VLSI technology continues its remarkable advances, the power consumption has been increased exponentially [2]. Current battery technology, with its 5% annual capacity increase [2], cannot effectively address this problem. In addition, serious concerns are raised with regards to managing the heat dissipation from the rapidly elevated power consumption. Left unchecked, the

power consumption will threaten to curtail the availability of the future high performance portable devices and advanced multimedia functionality on these devices.

Power aware scheduling has been proven to be an effectively way to reduce the power consumption. Rooted in the traditional real-time scheduling technology, the power aware scheduling techniques change the system computing performance accordingly based on the dynamically varied computation demand. Two main types of power management mechanisms are reported in the literature. The first one is commonly known as the *dynamic power down* (DPD), i.e., to shut down a processing unit and save power when it is idle, and switch over to normal state when the idle time expired. The second one is to update the processor's supply voltages and working frequencies, which is usually referred as the *dynamic voltage scaling* (DVS). Thanks to the recent IC technology advancement, many modern processors can vary its voltage and working frequencies (and thus processing speeds) dynamically.

For the past few years, we have seen extensive researches that employ a variety of real-time scheduling techniques to reduce the energy consumption. Most of them have been focused solely on reducing the energy consumption by the processor (e.g. [22, 25, 34]). A typical portable device usually includes one or more core processors, memory, and peripheral devices such as network interface card, disk, and display. While the processor is one of the major power hungry units in the system, other peripherals such as network interface card, memory banks, disks also consume significant amount of power. The empirical study by Viredaz and Wallach reveals that the processor core consumes around 28.8% of total power when playing a video file on a hardware testbed [32] for handheld devices, while the DRAM consumes about 28.4% of the total power. Note that this testbed [32] lacks disk storage and wireless networking capability, which may contribute as much power consumption as the processor core if not more [35, 9]. This implies that the techniques that reducing the processor energy alone may not be overall energy efficient.

A number of techniques (e.g. [11, 13, 38]) have been published recently to deal with the power consumption for systems consisting of DVS processors and peripheral devices. A fundamental tradeoff that has to be made in these approaches is whether to apply DVS or DPD during the scheduling process. DVS techniques reduce energy consumption by lowering the processor speed. Unfortunately, it extends the execution times of real-time tasks and shrinks the idle intervals, which is not favorable for DPD. Kim and Ha [13] proposed a technique for *hard* real-time system, while scheduling decisions are made on a timeslot-by-timeslot basis. To facilitate a run-time mechanism, the processor speed for each task is determined by analyzing the energy savings based on a pre-determined set of execution times. Jejurikar and Gupta [11] introduced a heuristic search

method to slow down the processor speed and optimize the energy usage by both the processor and peripheral devices. Zhuo and Chakrabarti [38] proposed a theoretical formulation of the optimal scaling factor and computed it numerically. Based on this factor, they introduced a dynamic scheduling technique that reduces the potential excessive preemptions among tasks to further reduce the system wide energy consumption.

While DVS techniques can dramatically reduce the dynamic power consumption for the processor, DPD techniques seems to be more promising in reducing the system-wide overall energy consumption in the near future. As shown in the work by Zhuo and Chakrabarti [38], when peripheral devices consume more power than the processor, the effectiveness of DVS techniques can be seriously degraded. Even for the processor itself, the energy efficiency of DVS is becoming limited as IC technology continue its evolution [36], especially when the leakage power is increasing exponentially and will soon surpass the dynamic power consumption [10]. DPD, on the other hand, is one of the most intuitive and effective ways to control the leakage power consumption. Moreover, most peripheral devices do not support DVS at all. As a result, the research on employing DPD has regained its momentum to reduce the system-wide energy consumption.

As a traditional energy-saving technique, DPD has been widely adopted in real-time scheduling. A majority of DPD techniques (e.g. [28, 23]) have been proposed for soft real-time systems, where task deadlines can be missed albeit with reduced quality levels. There are also a number of papers (e.g. [7, 30, 31]) deal with the power optimization for hard real-time systems, where a deadline miss is considered a system failure. A good survey on DPM related real-time scheduling research can be found in [4, 3].

Few real-time applications are truly *hard* real-time, i.e., missing one task deadline does not necessarily crash the entire application or system. Many real-time applications, such as multimedia and communication applications, can often tolerate occasional deadline misses, but too much deadline misses cannot satisfy user's perceived quality of service (QoS) requirement. While the statistic information such as the average deadline miss rate is commonly used to quantify the system performance, this metric can be problematic. Note that even a very low average miss rate tolerance cannot prevent a large number of deadline misses from occurring in a very short period of time. This may cause the loss of critical information which cannot be reconstructed and therefore severely degrade the service quality from user's perspective.

The *weakly hard real-time model* is a more suitable real-time model for this type of applications. The weakly-hard real-time task has both a firm deadline (i.e., a task instance missing its deadline is utterly useless) and a throughput requirement (i.e., there should be *sufficient* task instances from the same task meeting deadlines in order to provide required quality

levels). Several weakly-hard models have been introduced [26, 8, 14, 5, 33]. Ramanathan *et. al.* [26] proposed a so-called $(m,k)-$model, with a periodic task being associated with a pair of integers, i.e., (m,k), such that among any k consecutive instances of the task, at least m of the instances must finish by their deadlines for the system behavior to be acceptable. A *dynamic failure* occurs, which implies that the temporal QoS constraint is violated and the scheduler is thus considered failed, if within any consecutive k jobs more than $(k-m)$ job instances miss their deadlines. Koren *et. al.* [14] proposed a 'skip-over' model, which is a special case of (m,k) model with $m = k-1$. West *et. al.* [33] introduced another similar model, called the *window-constrained* model, which requires that within any *non-overlapped* and *consecutive* windows each of which containing k jobs, at least m of them can meet their deadlines.

In this paper, we study the problem of employing DPD to reduce the system-wide energy consumption with guaranteed QoS for a weakly hard real-time system. Specifically, we adopt the (m,k)-model to capture the QoS requirement for the real-time application. A key challenge for this problem has to do with the definition of which jobs are mandatory, i.e., whose deadlines have to be met to guarantee no dynamic failure occur, and which jobs can be optional. This problem has shows to be NP-hard even without the considerations of the power conservation [24]. In our approach, we employ a run-time technique and dynamically choose and execute the mandatory jobs in such a way that facilitates the system shut down. Our experiments shows that by judiciously choosing and merging the mandatory jobs, our techniques can lead to significant energy savings while still *guaranteeing* the (m,k)-firm deadlines.

The rest of the paper is organized as follows. We first introduce the system model, background, and motivations for our research in Section 2. Section 3 describes a feasibility condition to guarantee the (m,k)-firm deadlines in our approach when dynamically determining the mandatory/optional jobs. Section 4 presents two methods to delay the job execution to further extend the idle interval. Section 5 discusses how to judiciously execute the optional jobs and present the overall algorithm. Section 6 presents our experimental results. Section 7 draws the conclusions.

2 Preliminary

In this section, we first introduce the system model and the related work on real-time scheduling with (m,k)-firm guarantee. We then present and discuss a motivation example.

2.1 System models

We model a real-time application with n independent periodic tasks, $\mathcal{T} = \{\tau_0, \tau_1, \cdots, \tau_{n-1}\}$, scheduled according to the earliest deadline first (EDF) policy, i.e., the scheme that can best utilize the processor [16]. Each task contains an infinite sequence of periodically arriving instances called *jobs*. We use J_{ij} to represent the jth job of task τ_i. Task τ_i is characterized using the following five parameters:

- T_i: the time between the arrivals of two jobs from same task, referred to as the *period*.
- D_i: the time by which each job of τ_i must be completed, referred to as the *deadline*. We assume that $D_i \leq T_i$.
- C_i: the maximum number of processor cycles needed to complete one job of τ_i without any interruption, referred to as the *worst case execution time*.
- $(m_i, k_i)(0 < m_i \leq k_i)$: the (m,k)-constraint for τ_i, requiring that, among any k_i consecutive jobs of τ_i, at least m_i jobs meet their deadlines.

The system architecture consists of two functional units: a core processor and a peripheral device. Both the processor and the peripheral device can be shut down and waken up later when idle time expired. We denote the processor power with P_{pact} when running a task, and P_{pidle} when the processor is idle (yet still *on*). When the processor is shut down, its power consumption is denoted as P_{psleep}. The peripheral device in our system can be in one of two states: *active* or *sleep*. When the processor is active, the peripheral devices must be also in active mode to provide timely service. We assume that the device consumes the same power during its active mode no matter whether it is idle or not. The power consumption for the device is denoted as P_{dact} and P_{dsleep} for its active mode and sleep mode, respectively.

Time and energy needed to be consumed to shut-down and later wake up the processor and device. It will not be feasible or beneficial to shut down the system if the idle interval is not longer enough. We use T_{min} to represent the minimal idle intervals that can be feasibly shut-down with positive energy gains.

With the above system models, our problem can be formulated as follows:

Problem 1 *Given weakly hard real-time task set $\mathcal{T}$ and system architecture $\mathcal{A}$, schedule $\mathcal{T}$ with EDF on system $\mathcal{A}$ such that all (m,k)-constraints are guaranteed and the total energy consumption is minimized.*

2.2 Real-time scheduling with (m,k)-firm deadline

To schedule a real-time task set with (m,k)-firm deadline involves two sub-problems: (i) mandatory/optional partitioning problem, i.e., to determine if a job should be mandatory or optional, and (ii) scheduling problem, i.e., to schedule these jobs properly to guarantee their deadlines. As proven in [24], both problems are NP-hard problems. In what follows, we briefly introduce some related real-time scheduling results for (m,k)-firm guarantee. For ease of our explanation, we use *patterns* to denote the mandatory/optional partitions. A pattern is an infinite binary sequence associated with each task such that a job is mandatory if its corresponding bit is "1" and optional otherwise.

The mandatory/optional partition decision can be made off-line or on-line. Two known static mandatory/optional partitioning strategies are reported in literature. The first one is called *the deeply-red pattern* or *R-pattern*, which was proposed by Koren *et al.* [14]. According to this technique, let

$$\pi_{ij} = \begin{cases} 1 & 0 \leq j \bmod k_i < m_i \\ 0 & \text{otherwise} \end{cases} \qquad j = 0, 1, \cdots \tag{1}$$

Then job J_{ij} is market as mandatory if $\pi_{ij} = 1$, or optional otherwise. The second one is proposed by Ramanathan *et al.* [27] as follows.

$$\pi_{ij} = \begin{cases} 1 & \text{if } j = \lfloor \lceil \frac{j \times m_i}{k_i} \rceil \times \frac{k_i}{m_i} \rfloor \\ 0 & \text{otherwise} \end{cases} \qquad j = 0, 1, \cdots \tag{2}$$

The (m,k)-pattern defined with formula (2) has the property that mandatory jobs are marked evenly, and is therefore referred as the *evenly distributed pattern* (or *E-pattern*) [21].

The most significant advantage of applying static patterns is that they enable the application of theoretic real-time techniques to analyze system feasibility. Analytical schedulability results are available [27, 20] for both fixed-priority and EDF scheduling policies, based on either R-pattern or E-pattern. The problem, however, is its poor adaptivity in dealing with the run-time variations, which is inherent in many real-time applications.

Dynamic mandatory/optional partitioning, on the other hand, is more flexible and therefore can accommodate run-time variations more effectively. The problem is how to ensure the deadlines of all the mandatory jobs. A number of dynamic mandatory/optional partitioning heuristics are proposed (e.g. [26, 29, 1]) with no guarantee for the deadlines of mandatory

jobs at all. Currently, two dynamic techniques published can ensure the (m,k)-guarantee. Bernet *et. al.* [6] proposed a Bi-Modal Scheduler, which runs jobs at two modes: normal mode and panic mode. A task is first executed at the normal mode and promoted to the panic mode if the dynamic failure will occur if it stays in the normal mode. Niu *et. al.* [21] proposed to shift the E-pattern dynamically when an optional job meets its deadline.

2.3 The motivations

Our goal is to shut down the processor and device efficiently and guarantee the (m,k)-constraints in the mean time. Although the static (m,k)-patterns can guarantee the (m,k)-constraints, they usually lead to large number of short and scattered idle intervals. Figure1(a) shows the EDF schedule by determining mandatory jobs based on E-patterns for a task set with three periodic tasks. As shown in Figure1(a), the mandatory jobs are distributed evenly. This is advantages to schedule task sets with high utilizations but not to reduce the number of idle intervals. Note that there are as many as 11 idle intervals in time interval [0,96] in this schedule.

DPD mechanism is in favor of longer and fewer idle intervals. An intuitive idea to reduce the number of idle intervals is to assign mandatory jobs as close as possible. This seems to make the R-pattern assignment a better choice. However, as shown in Figure1(b), the EDF schedule based on R-patterns results in 12 idle intervals within the same time interval. The reasons are two folds. First, from equation (1), an R-pattern always marks the first m_i jobs as mandatory jobs. The mandatory jobs from different tasks are likely to overlap for the first "window" but not necessarily for the following windows due to the differences of k's and periods from different tasks. Second, even though mandatory jobs and the time intervals in which they are supposed to run are overlapped (e.g., see interval [0,15] in Figure 1(b)), idle intervals still exist due to the deadline and arrival constraints for the tasks.

Figure 1(c) presents a schedule that can cut the number of idle intervals to as small as 4. A small number of idle intervals usually means longer idle interval length. As a result, the energy overhead for shutting down the processor and device can be reduced. In addition, some idle intervals that previously cannot be shut down because they are too short can now be done so. This can transform to significant energy savings. A careful study of Figure 1(c) would reveal that such solution is obtained by employing an irregular mandatory/optional job pattern, i.e., neither E-pattern nor R-pattern, together with carefully delaying the execution of mandatory jobs. The challenges are then how to define appropriate mandatory jobs and how to delay the executions of these jobs effectively such that the idle intervals can be merged while the (m,k)-constraints can be guaranteed.

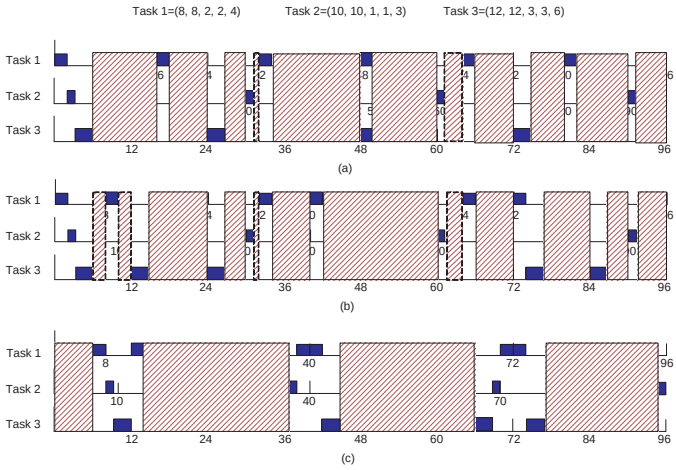

Figure 1. (a) The EDF schedule for three tasks according to E-patterns (with 11 idle intervals); (b) The EDF schedule for same tasks based on R-patterns (with 12 idle intervals); (c) A better schedule for the same task set (with only 4 idle intervals).

In following sections, we propose an integrated run-time technique to attack these challenges.

3 Meeting the (m,k)-constraints

From the motivation example shown above, it is evident that to the existing static (m,k)-patterns cannot effectively merge the idle intervals. How to devise new static (m,k)-patterns that can cluster mandatory jobs for this purpose is an interesting problem and needs further study. Nonetheless, the static patterns are usually based on worst case scenarios and less adaptive. Judiciously exploiting the variations, inevitable in the runtime environment, dynamically can be extremely beneficial. The problem is how to determine the patterns dynamically and ensure that no dynamic failure will ever occur. The following condition is critical in our approach when choosing mandatory jobs and ensure their feasibility.

Lemma 1 *Given system $\mathcal{T}$, let $\mathcal{M}$ be the mandatory job set according to their R-patterns. Then if $\mathcal{M}$ is EDF-schedulable, a job (i.e. J_p) can be marked as mandatory and meet its deadline if for each $\tau_i \in \mathcal{T}, i = 0,1,...,n-1$, no more than m_i jobs (including J_p) among any consecutive k_i jobs are marked as mandatory.*

Proof: For an arbitrary real-time task set, i.e., $\mathcal{T}$, scheduled with EDF, Zheng *et. al.* [37] and Liebeherr *et. al.* [15] showed that $\mathcal{T}$ is EDF-schedulable *iff*

$$\forall t > 0, \sum_i W_i(0,t) \leq t, \tag{3}$$

where $W_i(0,t)$ is the total workload from the jobs of τ_i that arrive before t and *must* be finished by t, or the so called *work demand*.

Given any task set schedulable with R-pattern and time t, let the mandatory workload within $[0,t)$ be $W(0,t)$. Then from equation (3) we have

$$\forall t > 0, W(0,t) \leq t. \tag{4}$$

In addition, from equation 1, we can see that there are *exactly* m_i jobs with any k_i consecutive jobs in $\mathcal{M}$. If we use $\mathcal{M}'$ to represent any other mandatory job sets in which no more than m_i jobs among any consecutive k_i jobs from τ_i are mandatory, and let its mandatory workload with $[0,t]$ be $\tilde{W}(0,t)$, then we must have $\tilde{W}(0,t) \leq W(0,t)$. Therefore, any mandatory job from $\mathcal{M}'$ can meet its deadline. □

Lemma 1 implies that as long as a task set is schedulable under R-patterns, we can flexibly choose a job as mandatory provided we do not choose more than m_i among k_i consecutive jobs from same task τ_i. Therefore, when the system is idle, we can intentionally delay the *assignment* of mandatory jobs in such a way that they can be congregated. However, recall that in the motivation example, even though the mandatory jobs are allocated closely, large number of idle intervals may still exist due to their arrival and deadline constraints. In next section, we solve this problem by carefully delaying the *execution* of mandatory jobs.

4 Delaying the execution of mandatory job set

When the processor is idle, delaying the execution of mandatory jobs helps to extend the idle intervals. However, it may also potentially cause mandatory jobs to miss their deadlines and thus cause dynamic failure. A number of papers published [7, 12] proposed to compute the job delay amount for a hard real-time task set based on its utilization factor. These approaches cannot be applied for real-time system with weakly hard real-time constraints since the famous condition, i.e., $U \leq 1$ is not necessary for a weakly hard real-time system to be feasible. In this section, we develop two sufficient conditions for delaying the execution of mandatory jobs as late as possible without causing any dynamic failure. (The proofs are provided at the Appendix section.) Before we introduce these sufficient conditions, we first introduce the following definition.

Definition 1 *Assume that $\mathcal{M}$ is the mandatory job sets from $\mathcal{T}$ according to R-pattern and schedulable, and let R_i be the*

worst case response time (i.e., the time from a job arrival to its finish). The delay factor *for τ_i (denoted as Y_i) is defined as*

$$Y_i = (D_i - R_i). \tag{5}$$

The worst case response time can be computed in a similar way as that in [18]. Since we only need to compute once for each task off-line, a more intuitive way is to scan through the interval from $[0, LCM(k_i T_i)]$, $i = 0, ..., n-1$ to find the worst case response time for each task. With Definition 5, our first sufficient condition is formulated in the following Theorem.

Theorem 1 *Let $\mathcal{M}$ be the mandatory job set such that no more than m_i mandatory jobs assigned for any k_i consecutive jobs from $\tau_i \in \mathcal{T}$. Assume that processor is idle at $t = t_0$, and let the arrival time for mandatory job J_i from τ_i immediately after t_0 be r_i. Then if the processor resumes its execution at*

$$T_{LS}(\mathcal{M}) = \min_i (r_i + Y_i), i = 0, 1, ..., n-1, \tag{6}$$

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

Theorem 1 allows us to determine the maximal delay for mandatory jobs based on worst case response time analysis, which is available off-line. The advantage of this approach is its small run-time overhead. Unfortunately, same as any other off-line strategy, it suffers the pessimistic estimation due to its assumption of the worst case scenario, as exemplified in Figure 2. Figure 2(a) shows the schedule of a task set of three tasks according to their static R-patterns. We can readily identify that $Y_1 = 4, Y_2 = 0$, and $Y_3 = 2$. Assume a dynamically determined mandatory job sets shown in Figure 2(b). (We can see that the job execution intervals are largely overlapped.) Since $Y_2 = 0$, the mandatory job from Task 2 cannot be delayed according to Theorem 1, and there is one idle interval between [28,36]. On the other hand, however, if we delay the processor execution till $t = 27$ (as shown in Figure 2(d)), all jobs can meet their deadline and no idle interval exists. This is because that Theorem 1 computes the maximal delay assuming the job always takes its worst case response time. When a job has a much smaller response time, it can be delayed further and may thus be more effective in reducing the idle intervals.

Mochocki *et. al.* [17] introduced a method to compute the latest starting time (LST) for a real-time job set. Their method is based on the following lemma.

Lemma 2 *[17] Let job set $\mathcal{J}$=$\{J_0, J_1, ..., J_s\}$ and $J_i = \{r_i, d_i, c_i\}$, where r_i, d_i, and c_i refer to the arrival time, deadline, and*

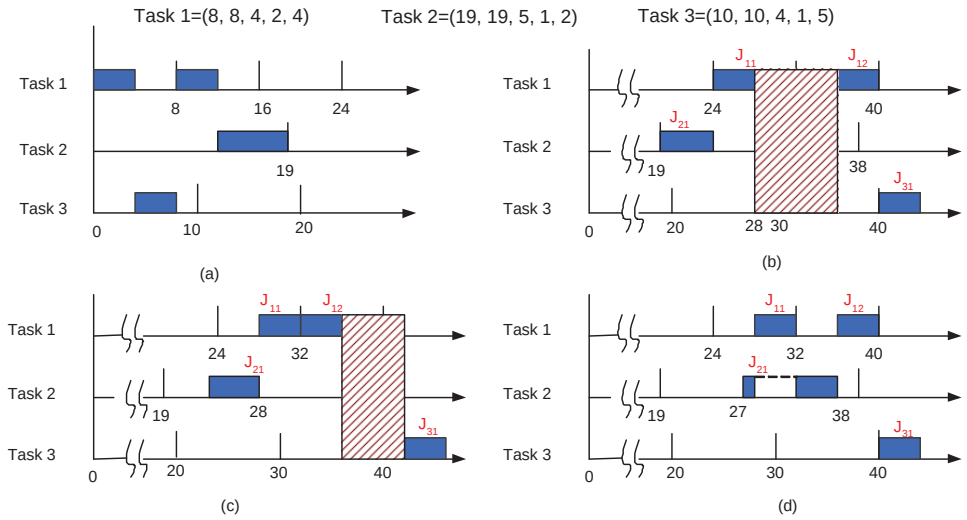

Figure 2. (a) Three tasks scheduled based on their R-Pattern; (b) J_{21} cannot be delayed according to Theorem 1; (c) Delaying the mandatory jobs to $t = 23$ ([]) cannot remove the idle interval; (d) Delaying the mandatory jobs to $t = 27$ and eliminating the idle interval.

execution time of J_i, respectively. Let

$$t_{LS}(J_i) = d_i - \sum_{J_k \in hp(J_i)} c_k, \tag{7}$$

where $hp(J_k)$ is the jobs with the same or higher priorities than that of J_k. Then the latest starting time (LST) of $\mathcal{J}$, i.e., $T_{LS}(\mathcal{J})$, without violating deadline constraints is

$$T_{LS}(\mathcal{J}) = \min_i t_{LS}(J_i). \tag{8}$$

Lemma 2 helps to compute the LST for a given job set. However, this method cannot be readily applied in our dynamic approach where the job set is not statically determined. Niu *et al.* [19] later extended Lemma 2 and compute LST based on information from only a subset of the jobs. This approach has a much lower complexity and hence is more suitable for on-line purpose. We use Figure 2(c) to illustrate this approach.

Assume the processor is idle before $t = 19$ in Figure 2(c). Since the LST for a job set is bounded by the earliest deadline of the jobs (so called *delay bound* and denoted as T_B), and is usually known on-line (i.e. $T_B = 32$ in this case), it is desirable to estimate LST for the entire job set based on the jobs arriving before the delay bound,i.e., J_{11} and J_{21}. As pointed out in [19], the LST computed by employing equation (7) directly for J_{11} and J_{21} may not be valid since the validity of LST in Lemma 2

is ensured by employing (7) for *every* job in the job set. 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 (7). 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 coming low priority job, whichever is smaller. In Figure 2(c), both the effective deadlines for J_{11} and J_{21} happen to be 32. Therefore, based on equation (8), $T_{LS} = \min(t_{LS}(J_{11}), t_{LS}(J_{11})) = 23$.

The approach in [19] delays mandatory jobs further than the one applying Theorem 1 and shorten the idle interval in Figure 2(b). However, it fails to eliminate the idle interval. In what follows, we present another method to estimate the LST for mandatory jobs. Our method maintains the same computational complexity as that in [19] but with a more accurate estimation. Specifically, our method is formally formulated in Theorem 2.

Theorem 2 *Let $\mathcal{M}$ be the mandatory job set such that no more than m_i mandatory jobs assigned for any k_i consecutive jobs from $\tau_i \in \mathcal{T}$. Assume that processor is idle at $t = t_0$, and let the delay bound (i.e., the earliest deadline for the coming mandatory jobs) be T_B for $\mathcal{M}$. Then no mandatory job in $\mathcal{M}$ will miss its deadline if the processor resumes its execution at $\tilde{T}_{LS}(\mathcal{M})$, where*

$$\tilde{T}_{LS}(\mathcal{M}) = \min_{J_i \in \mathcal{J}_s}(d_i^* - \sum_{J_k \in hp(J_i)} c_k), \tag{9}$$

where $\mathcal{J}_s$ consists of mandatory jobs from $\mathcal{M}$ with arrival times earlier than T_B but later than t_0, and

$$d_i^* = \min_p(d_i, r_p + Y_p), \forall J_p \in \mathcal{M}, J_p \notin \mathcal{J}_s \text{ and } d_p > d_i. \tag{10}$$

The fundamental difference between our technique and the one in [19] is the way that effective deadlines are defined. From equation (10) in Theorem 2, the effective deadline for a mandatory job is relaxed from the earliest arrival time of the next lower priority job further with its delay factor. This in turn will allow mandatory jobs to delay further to merge the idle interval. As such, the effective deadline for J_{21} becomes 34 instead of 32, and thus we have $\tilde{T}_{LS} = 27$, which is the case shown in Figure 2(d). Note that, since Y_i is available off-line, our technique based on Theorem 2 has the same on-line complexity as that in [19]. Also, it is not difficult that the LST computed based on Theorem 2 is never worst than that by the technique in [19]. Finally, it is worthy to mention that both Theorem 1 and Theorem 2 are sufficient conditions. Therefore, the larger one from equation (6 and (9) can be used as LST and guarantee the deadlines for all the mandatory jobs.

5 Executing of the optional jobs

When the system are idle, Lemma 1 helps us to *assign* a mandatory job as late as possible, and Theorem 1 and Theorem 2 can further delay the idle intervals by delaying the execution of mandatory jobs. When the predicted idle interval is long enough (i.e. greater than T_{min}), it will be beneficial to shut down the processor and devices. One missing piece in our approach is, however, what if the idle interval is still not long enough?

We have two choices when the idle interval is not long enough to accommodate the timing and energy overhead: (1) we can simply keep the system idle (but active); (2) we can opt to run some optional jobs. For the first case, the processor consumes a little less power (as $P_{pact} < P_{pidle}$) while the device consumes nearly the same power. At the first sight, running optional jobs does not seem to be energy efficient since $P_{pact} > P_{pidle}$. However, executing optional jobs may potentially lead to positive energy saving gain because (1) some mandatory jobs become optional and do not have to be executed; and (2) more importantly, some short idle intervals in the future can be merged to longer ones and enable system to shut down if appropriate mandatory jobs are demoted to optional. The problem is how to select the *right* optional jobs.

To make a precise analysis of the trade off in executing the optional jobs is a challenging problem, especially from the dynamic scheduling perspective. In considering this, we resort to a heuristic approach in solving this problem. In our heuristic approach, an optional job is executed, non-preemptively, only when it can finish within the idle intervals as predicted. This helps to avoid the execution of too many optional jobs, which would not be energy efficient. When there are more than one candidate optional jobs, we devise a function to evaluate the fitness of an optional job. The fitness function, i.e. $\mathcal{F}$, is determined by two parameters, i.e., the flexibility (F) and criticality (Cr). An optional tends to have higher energy-saving potential if its corresponding mandatory jobs are more flexible to be moved around and/or it is closer to dynamic failure. Therefore, for optional job J_{ij}, we define

$$F(J_{ij}) = (Y_i + D_i) \times \frac{k_i T_i}{m_i C_i}, \tag{11}$$

and

$$Cr(J_{ij}) = \frac{m_i'}{k_i - m_i}, \tag{12}$$

where m_i' is the currently allowed deadline misses of τ_i without causing dynamic failure. Note that $F(J_{ij})$ can be computed off-line but $C(J_{ij})$ is computed on-line.

The rationale behind equation (11) is that, from Theorem 1 and Theorem 2, large Y_i and D_i tend to make future mandatory jobs from τ_i more flexible to be delayed. On the other hand, $\frac{m_i C_i}{k_i T_i}$ indicates the average mandatory workload for task τ_i. The higher the value is, the more difficult it is to shift the workload and thus merge idle intervals. Equation (12) measures the number (normalized) of deadline misses that can still be tolerated. The higher the value, the less urgent that J_{ij} needs to be executed in order not to cause a dynamic failure. Note that if J_{ij} is optional, $C(J_{ij})$ cannot be zero. Therefore, based on equation (11) and (12, we define $\mathcal{F}$ as

$$\mathcal{F}(J_{ij}) = \frac{F^*(\tau_i)}{Cr(J_{ij})}, \tag{13}$$

where $F^*(\tau_i)$ is the normalized value of $F(J_{ij})$ (based on the largest value) for consistency.

Now, with our heuristic to choose proper optional jobs introduced above, we are ready to present our overall algorithm (Algorithm 1) for Problem 1. The algorithm consists of two phases: an off-line phase and an on-line phase. During the off-line phase, the worst case response time for each task under its *R*-pattern is computed. At the same time, the delay factor, i.e., Y_i and $F^*(\tau_i)$ are also computed for later on-line use (line 3). During on-line phase, we keep track of the history of a task and assign a job to be mandatory if it misses deadline will cause a dynamic failure (line 5). When processor is not idle, we execute the mandatory job as soon as possible according to EDF schedule (line 7). If the processor is idle, we delay the execution of mandatory job as late as possible to extend the idle intervals (line 10). If the idle interval is large enough, we shut down the system and later wake up the system when the idle time expired (line 13). Otherwise, we execute proper candidate optional job based on their fitness values (line 15-17).

The feasibility of Algorithm 1 is guaranteed by Lemma 1, Theorem 1, and Theorem 2. Note that exactly m_i out of k_i from a task τ_i are chosen as mandatory, but not all of them are executed since some optional jobs from the same task may have been executed. On the other hand, the optional jobs are only executed when the processor is "idle" when executing the mandatory jobs. Therefore, as long as a task set is schedulable with static R-pattern, our algorithm always guarantees that no dynamic failure will ever occur. Further, the energy efficiency of our dynamic approach lies in the fact that it adjusts the mandatory/optional partition adaptively by incorporating the run-time information and merging smaller idle intervals into larger ones. It is particularly efficient considering the fact that the actual execution time of a task can be much smaller than its worst case execution time. In the next section, we use experiments to evaluate the performance of our algorithm.

Algorithm 1 The overall algorithm.

Input: $\mathcal{T}$ and T_{min}.
Offline:
Compute R_i, Y_i, $F^*(\tau_i)$ for $\tau_i \in \mathcal{T}$;
Online:
A job is marked as mandatory if a dynamic failure will occur if it fails to meet its deadline;
if processor is not idle **then**
  Run mandatory jobs according to EDF;
else
  Let $\mathcal{J}$ be the coming mandatory job set;
  Compute the maximal delay T_{LS} for $\mathcal{J}$ based on Theorem 1 and Theorem 2;
  if $T_{LS} - t_{cur} > T_{min}$ **then**
    // t_{cur} is the current time
    Shut down the processor and set up the wake up timer to be $T_{LS} - t_{cur}$;
  else
    $\mathcal{J}_o =$ the optional jobs in the ready queue;
    Compute the fitness based on equation (13).
    Run $J_i \in \mathcal{J}_o$ non-preemptively that have the maximum fitness value;
  end if
end if

6 Experiments

In this section, we evaluate the performance of our approach using simulations. We implemented five approaches in our experiments. In the first approach, the mandatory jobs were statically determined using the *R*-patterns. We refer this approach as DPD_R and use its results as the reference results. The second approach also performed the mandatory/optional partitioning statically. Different from DPD_R, we used E-pattern instead of R-pattern in this approach and hereby refer it as DPD_E. In the third approach, we marked the mandatory jobs dynamically as described in Algorithm 1. However, the execution of the mandatory jobs were not delayed. We refer this approach as DPD_{ND}. The fourth approach also determined the mandatory job dynamically and delayed the mandatory job executions. The delay amount is computed based on the approach in [19]. We call this approach as DPD_{NTA}. The final approach, denoted by DPD_{DYN}, is our new approach presented in this paper, i.e., the complete implementation of Algorithm 1.

The periodic task set in our experiments consisted of five tasks. Each task set were randomly generated with the periods randomly chosen in the range of $[10, 50]ms$. We assumed that the deadlines for the tasks were the same as their periods. The worst case execution time (WCET) of a task was set to be uniformly distributed from $1ms$ to its deadline, and we assumed that the actual execution time for a job was evenly distributed from [0.4WCET, 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 $1 \leq m_i < k_i$. To investigate the

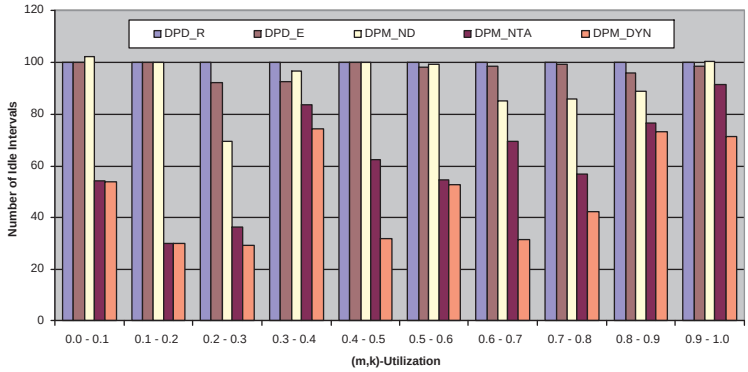

Figure 3. The average number of idle intervals by different approaches.

performance for different approaches under different workload, we divided the total (m,k)-utilization, i.e., $\sum_i \frac{m_i C_i}{k_i T_i}$, into intervals of length 0.1. To reduce the statistical errors, we require that each interval contain at least 20 schedulable task sets, or at least 5000 task sets within each interval have been generated. For the processor and device considered in our experiments, we assume that $P_{pact} = 1.0W$, $P_{pidle} = \frac{1}{3}P_{pact}$, and $P_{dact} = 1.0W$. We assume that the power consumption for the processor and device during the sleep mode are negligible. We also assume the minimal idle interval length to be $3ms$.

We first study the number of idle intervals by the five different scheduling strategies. A large number of idle intervals is undesirable in DPD since it either incurs higher transition overhead due to more frequent transitions or has to keep system busy due to shorter idle interval lengths. Figure 3 compares the normalized (with respect to DPD_R) number of idle intervals within LCM($k_i T_i$) by different approaches.

Figure 3 clearly shows that our proposed technique (i.e. DPD_DYN) can dramatically reduce the number of idle intervals. It is interesting to see that the numbers of idle intervals by DPD_E and DPD_R are quite close which shows that both static approaches are not effective in merging the idle intervals. When compared with our approach, the number of idle intervals by DPD_R and DPD_E can be as nearly 3.5 times higher than our approach. In addition, we can observe from Figure 3 that, if we only dynamically change the mandatory job assignment without delaying the execution of the mandatory jobs (i.e. $DPD_N D$, it may help to merge the idle intervals in some cases but not always. And the result number of idle intervals is still much larger than those that delaying the processor execution. Furthermore, compared with DPD_{NTA}, i.e., the approach that adopts a different way to delay the mandatory jobs [19], our approach can cut its idle interval number as many as a half. This results

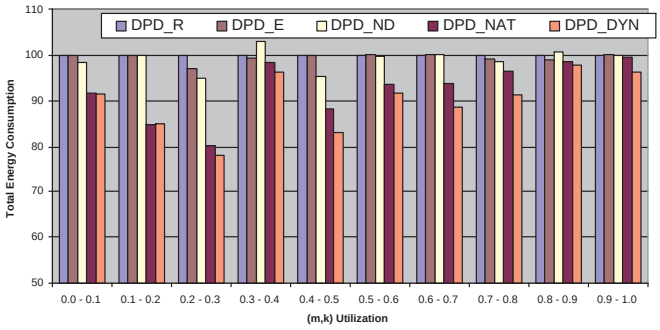

Figure 4. The average total energy consumption by different approaches.

demonstrates favorably the strength of the two sufficient conditions presented in section 4.

The reduction of idle intervals has a strong correlation with the reduction of energy as can be shown in Figure 4. Figure 4 illustrates the overall energy consumption for the same task sets by different approaches. From Figure 4, it is not surprising to see that the energy savings obtained with DPD_{DYN} varies according to the (m,k)-utilization. When (m,k)-utilization is very high (e.g. within [0.8,1.0]), the system is busy most of the time and cannot be shut down. Under this scenarios, all the approaches have the similar energy savings. When the (m,k)-utilization is small, we can see that DPD_{DYN} can save energy more effectively. As shown in Figure 4, DPD_{DYN} can reduce the energy consumption of DPD_ND by up to 18%, and can that of DPD_NTA up to 6% *without* increasing the on-line complexity. The energy conservation is more significant when compared with the conventional and naive approaches (DPP_E and DPP_R), i.e., up to over 23%. In summary, the experiment results has shown that our approach can significantly reduce the idle intervals, and hence achieve better energy savings with guaranteed QoS level that the conventional approaches.

7 Conclusions

Energy consumption is critical in the design of pervasive real-time computing platforms. The power consumption for peripheral devices, as a significant part of the overall power consumption, must be taken into consideration to reduce the system wide power consumption. On the other hand, most of these real-time systems are not hard real-time but exhibit more complex QoS behaviors that can only be modeled by more complicated constraints. In this paper, we presented a dynamic DPD approach to reduce the system wide energy consumption while guaranteeing the QoS requirement, which are modeled

as the (m,k)-constraints. Our approach ensures the (m,k)-firm guarantee by taking the advantage of static analysis. The energy saving performance of our approach comes from the facts that we dynamically change the mandatory/optional job settings, and merge the idle intervals effectively by delaying the execution for mandatory jobs. Our experimental results demonstrate that our approach can greatly reduce the number of idle intervals and thus the power consumption, while still providing (m,k)-firm guarantee.

References

[1] T. A. AlEnawy and H. Aydin. Energy-constrained scheduling for weakly-hard real-time systems. *Real-Time Systems Symposium*, pages 376–385, 2005.

[2] T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabarti, and W. Wolf. Mobile supercomputers. *IEEE Computer*, 37(5):81–83, 2004.

[3] L. Benini, A. Bogliolo, and G. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Trans. on VLSI*, 8(3):299–316, June 2000.

[4] L. Benini and G. Micheli. *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer, 1997.

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

[6] G. Bernat and R. Cayssials. Guarantted on-line weakly-hard real-time systems. In *RTSS*, 2001.

[7] H. Cheng and S. Goddard. Online energy-aware i/o device scheduling for hard real-time systems. *International conference on Design, automation and test in Europe*, pages 1055–1060, 2006.

[8] J.-Y. Chung, J. W. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Trans. on Computers*, 39(9):1156–1175, September 1990.

[9] L. Doherty, B. Warneke, B. Boser, and K. Pister. Energy and performance considerations for smart dust. *International Journal of Parallel Distributed Systems and Networks*, 4(3):121–133, 2001.

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

[11] R. Jejurikar and R. Gupta. Dynamic voltage scaling for system-wide energy minimization in real-time embedded systems. *International Symposium on Low Power Electronics and Design*, pages 78–81, 2004.

[12] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. *DAC*, pages 275 – 280, 2004.

[13] M. Kim and S. Ha. Hybrid run-time power management technique for real-time embedded system with voltage scalable processor. *ACM SIGPLAN workshop on Optimization of middleware and distributed systems*, pages 11–19, 2001.

[14] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *RTSS*, 1995.

[15] J. Liebeherr, D. Wrege, and D. Ferrari. Exact admission control for networks with a bounded delay service. *IEEE Trans. on Networking*, 4(6):885–901, 1996.

[16] 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.

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

[18] M.Spuri. Analysis of deadline scheduled real-time systems. In *Rapport de Recherche RR-2772, INRIA, France*, 1996.

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

[20] L. Niu and G. Quan. Energy-aware scheduling for realtime systems with (m,k)-guarantee. *Technical Report TR-2005-05, Department of Computer Science and Engineering, University of South Carolina*, 2005.

[21] L. Niu and G. Quan. A hybrid static/dynamic dvs scheduling for real-time systems with (m, k)-guarantee. *Real-Time Systems Symposium*, pages 356–365, 2005.

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

[23] Q. Qiu, Q. Wu, and M.Pedram. Dynamic power management in a mobile multimedia system with guaranteed quality-of-service. In *DAC*, pages 834–839, 2001.

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

[25] G. Quan and X. S. Hu. Energy efficient fixed-priority scheduling for real-time systems on voltage variable processors. In *DAC*, pages 828–833, 2001.

[26] K. Ramamritham and J. A. Stankovic. Scheduling algorithms and operating system support for real-time systems. *Proceedings of the IEEE*, 82(1):55–67, January 1994.

[27] 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.

[28] P. Rong and M. Pedram. Hierarchical power management with application to scheduling. *International symposium on Low power electronics and design*, pages 269–274, 2005.

[29] A. Striegel and G. Manimaran. Best-effort scheduling of (m,k)-firm real-time streams in multihop networks. *Workshop on Parallel and Distributed Real-Time Systems*, 2000.

[30] V. Swaminathan and K. Chakrabarty. Energy-conscious, deterministic i/o device scheduling in hard real-time systems. *IEEE Trans. on CAD*, 22(7):847–858, 2003.

[31] V. Swaminathan and K. Chakrabarty. Pruning-based, energy-optimal, deterministic i/o device scheduling for hard real-time systems. *Trans. on Embedded Computing Sys.*, 4(1):141–167, 2005.

[32] M. A. Viredaz and D. A. Wallach. Power evaluation of a handheld computer. *IEEE Micro*, 23(1):66–74, 2003.

[33] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. In *ICMCS*, 1999.

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

[35] J. Zedlewski, S. Sobti, N. Garg, F. Zheng, A. Krishnamurthy, and R. Wang. Modeling hard-disk power consumption. *2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 217–230, 2003.

[36] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner. Theoretical and practical limits of dynamic voltage scaling. *DAC*, pages 868–873, 2004.

[37] Q. Zheng and K. G. Shin. On the ability of establishing real-time channels in point-to-point packet-switched networks. *IEEE Trans. on Comm.*, 42(2/3/4):1096–1105, 1994.

[38] J. Zhuo and C. Chakrabarti. Systemlevel energyefficient dynamic task scheduling. *Design Automation Conference*, pages 628–631, 2005.

8 Appendix

8.1 Proof for Theorem 1

To prove Theorem 1, we first prove the following lemma.

Lemma 3 *Let $\mathcal{M}$ be the mandatory job set from $\mathcal{T}$ according to the R-pattern. Then if the processor starts its execution at $t_s = \min(Y_i), i = 0,1,...,n-1$, no mandatory job in $\mathcal{M}$ will miss its deadline.*

Proof: Use contradiction. Assume that when the processor starts its execution at t_s, some mandatory job $J_p = /r_p, c_p, d_p/$ misses its deadline, where r_p, c_p, and d_p represent the arrival time, execution time, and absolute deadline of J_p. J_p must be in the first busy interval since the processor delay its execution would not cause J_p to miss its deadline otherwise. Therefore

$$\sum_i W_i(0,d_p) > (d_p - t_s). \tag{14}$$

where $W_i(t_1,t_2)$ represents the *work demand* (see the proof for Lemma 1) from $\mathcal{M}$ between interval $[t_1,t_2]$.

Assume J_p finishes at $f_p (r_p < f_p \leq d_p)$ when the processor starts at $t = 0$. Let

- $\mathcal{J}(0,d_p)$ represent the mandatory jobs with deadlines no later than d_p;

- $\mathcal{J}(0, d_f)$ represent the mandatory jobs arriving earlier than f_p with deadlines no later than than d_p;
- $\mathcal{J}(d_f, d_p)$ represent the mandatory jobs arriving *NO* earlier than f_p with deadlines no later than d_p.

It is easy to see that $\mathcal{J}(0, d_p) = \mathcal{J}(0, d_f) \cup \mathcal{J}(d_f, d_p)$. Let $W'(\mathcal{J})$ represent the workload, i.e., total execution time, of $\mathcal{J}$. Then we have

$$\sum_i W_i(0, d_p) = W'(\mathcal{J}(0, d_p)) = W'(\mathcal{J}(0, f_p)) + W'(\mathcal{J}(f_p, d_p)). \tag{15}$$

and

$$W'(\mathcal{J}(0, f_p)) \leq f_p. \tag{16}$$

Now consider job $J_q = r_q, c_q, d_q \in \mathcal{J}(d_f, d_p)$ such that J_q finishes at f_q (the latest before d_p) when the processor starts at $t = 0$. Then we have

$$W'(\mathcal{J}(f_p, d_p)) \leq (d_p - f_p) - (d_q - f_q). \tag{17}$$

As $(d_q - f_q) \geq (D_q - R_q) = Y_q$ (Definition 5) and $Y_q \geq t_s$, from equation (17), we have

$$W'(\mathcal{J}(f_p, d_p)) \leq (d_p - f_p) - t_s. \tag{18}$$

Then from equation (15, (16), and (18), we have

$$\sum_i W_i(0, d_p) \leq d_p - t_s, \tag{19}$$

which contradicts equation (14). □

We now proceed to prove Theorem 1. Again we use contradiction. Assume the processor is idle at t_0 and a mandatory job J_p misses its deadline when the processor resumes its execution at $t = t_0 + t_s$. Therefore we have

$$\sum_i W_i(t_0, d_p) > d_p - t. \tag{20}$$

Now consider $\mathcal{M}'$, the mandatory job set from $\mathcal{T}$ according to the R-pattern. Since $\mathcal{M}'$ is schedulable when processor delays its execution to t_s, we have

$$\sum_i W_i(0, d_p - t_0) < d_p - t_0 - t_s = d_p - t. \tag{21}$$

In addition, for $\mathcal{M}$, there are no more than m_i jobs among any consecutive k_i jobs from τ_i are mandatory. Therefore, we have

$$\sum_i W_i(0, d_p - t_0) \geq \sum_i W_i(t_0, d_p) > d_p - t, \tag{22}$$

which contradicts equation (20).

8.2 Proof for Theorem 2

We prove Theorem 2 based on Theorem 1. Assume the processor resumes its execution at $\tilde{T}_{LS}(\mathcal{M})$, as defined in equation (9). Let J_p be the first mandatory that is executed. From Lemma 2, it is easy to see that J_p and all higher priority jobs are schedulable. For any job J_i with priority lower than that of J_n, we consider two cases: (a) $r_i < T_B$, (b) $r_i \geq T_B$. When $r_i < T_B$, similarly as J_p, its schedulability is guaranteed. When $r_i \geq T_B$, note that the jobs are delayed no more than $\min_i r_i + Y_i$ and are therefore schedulable according to Theorem 1. Thus, all the mandatory jobs can meet their deadlines.