Fault-Tolerant Energy Management for Real-Time Systems with Weakly Hard QoS Assurance

Linwei Niu
Department of Electrical Engineering and Computer Science
Howard University
Washington, DC, U.S.A.
linwei.niu@howard.edu

***Abstract*—While energy consumption is the primary concern for the design of real-time embedded systems, fault-tolerance and quality of service (QoS) are becoming increasingly important in the development of today's pervasive computing systems. In this work, we study the problem of energy-aware standby-sparing for weakly hard real-time embedded systems. The standby-sparing systems adopt a primary processor and a spare processor to provide fault tolerance for both permanent and transient faults. In order to reduce energy consumption for such kind of systems, we proposed two novel scheduling schemes: one for $(1,1)$-hard tasks and one for general (m,k)-hard tasks which require that at least m out of any k consecutive jobs of a task meet their deadlines. Through extensive evaluations, our results demonstrate that the proposed techniques significantly outperform the previous research in reducing energy consumption for both $(1,1)$-hard task sets and general (m,k)-hard task sets while assuring fault tolerance through standby-sparing.**

***Index Terms*—energy efficiency, fault tolerance, standby-sparing, QoS, real-time systems**

I. Introduction

With the advance of IC technology, energy efficiency has been a critical factor in the design of real-time embedded systems. On the other hand, fault tolerance has also been a major concern for pervasive computing systems as system fault(s) could occur anytime [1]. Generally, computing system faults can be classified into permanent faults and transient faults [2]. Permanent faults could be caused by hardware failure or permanent damage in processing unit(s) whereas transient faults are mainly due to transient factors such as electromagnetic interference and/or cosmic ray radiations.

Recently a lot of researches (e.g. [3], [4]) have been conducted on improving the energy efficiency for fault-tolerant real-time systems. Many of them have focused on dealing with transient faults. A widely adopted strategy is to use software redundancy, *i.e.*, to reserve recovery jobs, whenever possible, for the jobs subject to transient faults. For mission critical applications such as nuclear plant control systems, permanent faults need to be dealt with by all means to avoid system failure. Otherwise catastrophic consequences could occur. More recently, solutions adopting hardware redundancy are proposed to address this issue. Among them the *standby-sparing* technique has gained much attention [5]–[8]. Generally, the standby-sparing makes use of the redundancy of processing units in multicore/multiprocessor systems. More specifically, a standby-sparing system consists of two processors, a primary one and a spare one, executing in parallel. For each real-time job executed in the primary processor, there is a corresponding backup job reserved for it in the spare processor [7]. As such, whenever a permanent fault occurs to the primary or the spare processor, the other one can still continue without causing system failure. Moreover, it is not hard to see that the backup tasks/jobs in the spare processor can also help tolerate transient faults for their corresponding main tasks/jobs in the primary processor.

In a standby-sparing system, the execution of the main jobs in the primary processor and their corresponding backup jobs in the spare processor might need to be overlapped with each other. Thus the total energy consumption could be quite considerable. Regarding that, some recent works have been reported to reduce energy (e.g. [5]–[8]). The main idea is to try to let the executions of the main jobs and their corresponding backup jobs be shifted away as much as possible such that, once the main jobs are completed successfully, their corresponding backup jobs could be canceled early, thereby saving energy. With that in mind, in [6], [7], approaches based on EDL (earliest deadline as late as possible) scheme [9] were proposed for standby-sparing real-time systems. Their works are mainly focused on *hard* real-time systems, *i.e.*, the systems which require all real-time tasks/jobs meet their deadlines. However, in practical time-sensitive applications, such as multimedia or time-critical communication systems, occasional deadline misses are acceptable so long as the user perceived quality of service (QoS) can be ensured at certain levels. For such kind of systems, the existing techniques solely based on hard real-time constraints are insufficient in dealing with energy reduction under standby-sparing and more advanced techniques incorporating the QoS systematically are desired. To achieve this goal, the QoS requirements need to be quantified in certain ways. One popular existing approach is to use some statistic information such as the average deadline miss rate as the QoS metric. Although such kind of metric can ensure the quality of service in a probabilistic manner, it can still 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 consecutively in such a short period of time that the data cannot be successfully reconstructed [10].

The *weakly hard QoS model* is more appropriate to model such kind of systems. Under the weakly hard QoS model, tasks have both firm deadlines (i.e., task(s) with deadline(s) missed generate(s) no useful values) and a throughput requirement (i.e., *sufficient* task instances must finish before their deadlines to provide acceptable QoS levels) [11]. Two well known weakly hard QoS models are the (m,k)-model [12] and the *window-constrained* model [13]. The (m,k)-model requires that m jobs out of any *sliding* window of k consecutive jobs of the task meet their deadlines, whereas the *window-constrained* model requires that m jobs out of each *fixed* and *nonoverlapped* window of k consecutive jobs meet their deadlines. It is not hard to see that the *window-constrained* model is weaker than the (m,k)-model as the latter one is more restrictive.

To ensure the (m,k)-constraints, Ramanathan *et al.* [14] adopted a partitioning strategy which divides the jobs into *mandatory* and *optional* ones. The mandatory ones are the jobs that must meet their deadlines in order to satisfy the (m,k)-constraints. In other words, so long as all the mandatory jobs can meet their deadlines, the (m,k)-constraints can be ensured. In [13], West *et al.* tried to set up a correspondence relationship between the *window-constrained* model and the (m,k) model. They found that the *window-constraints* can be converted to the (m,k)-constraints through certain automatic way.

In this paper, we study the problem of reducing the energy consumption for fault-tolerant weakly hard real-time embedded systems using standby-sparing.

The rest of the paper is organized as follows. Section II presents the preliminaries. Section III presents our approach for $(1,1)$-hard tasks. Section IV presents our approach for general (m,k)-hard tasks. In Section V and Section VI, we present our evaluation results and conclusions.

II. Preliminaries

A. System models

The real-time system considered in this paper contains n independent periodic tasks, $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_N\}$, scheduled according to the earliest deadline first (EDF) scheduling scheme. Each task contains an infinite sequence of periodically arriving instances called *jobs*. Task τ_i is characterized using five parameters, *i.e.*, (C_i, D_i, P_i, m_i, k_i). C_i, D_i ($\leq P_i$), and P_i represent the worst case execution time (WCET), deadline, and period for τ_i, respectively. A pair of integers, *i.e.*, (m_i,k_i) $(0 < m_i \leq k_i)$, are used to represent the (m,k)-constraint for task τ_i which requires that, among any k_i consecutive jobs, at least m_i jobs are executed successfully. The j^{th} job of task τ_i is represented with J_{ij} and we use r_{ij}, $c_{ij}(=C_i)$, and d_{ij} to represent its release time, execution time, and absolute deadline, respectively.

The system consists of two identical processors which are denoted as primary processor and spare processor, respectively. For the purpose of tolerating permanent/transient faults, each mandatory job of a task τ_i has two duplicate copies running in the primary and the spare processors separately. Whenever a permanent fault is encountered in either processor, the other one will take over the whole system (to continue as normal). For convenience, we call each task τ_i (or mandatory job J_{ij}) running in the primary processor *main task/job* and its corresponding copy running in the spare processor *backup task/job*, denoted as $\tau_i^{'}$ (or $J_{ij}^{'}$).

The processor power when running a job is denoted as P_{act} which consists of both dynamic power and static power. Although dynamic power can be reduced effectively by dynamic voltage/frequency scaling (DVFS) techniques, the efficiency of DVFS in reducing the overall energy is becoming seriously degraded with the dramatic increase in static power (mainly due to leakage) with the shrinking of IC technology size. Dynamic power down (DPD), on the other hand, is more effective in controlling the static power when the processor is not in use. With that in mind, in this paper we assume that, when the processor is busy, it always consumes P_{act}. Without loss of generality, we normalize P_{act} to 1 and assume that one unit of energy will be consumed for a processor to execute a job for one time unit. When no job is pending for execution, the processors can be put into low-power state with DPD if the idle interval length is larger than the break even time T_{be} [7].

B. Fault Model

Similar to the standby-sparing systems in [6], [7], the system we considered can tolerate both permanent and transient faults. With the redundancy of the processing units, our system can tolerate at least one permanent fault in the primary or the spare processor. For transient faults which can occur anytime during the task execution, we assume they can be detected at the end of a job's execution using sanity (or consistency) checks [15] and the overhead for detection can be integrated into the job's execution time. Whenever a main job encounters transient fault(s), its backup job needs to be executed to completion.

With the above system and fault models, in the following we first show how to reduce energy consumption for $(1,1)$-hard, *i.e.* periodic hard real-time task sets. After that, we will explore how to deal with energy reduction for general (m,k)-hard task sets.

III. Standby-sparing for $(1,1)$-hard task sets

For $(1,1)$-hard task sets (or task sets in which $m_i = k_i$ for all tasks), under standby sparing, all jobs need to have two duplicate copies running in the primary and the spare processors, respectively. It is not hard to see that, due to the overlapped executions between them, one way to save energy is to let each main job in the primary processor be executed as soon as possible and its backup job in the spare processor be executed as late as possible such that, once the main job is completed successfully, its backup job can be canceled immediately, therefore saving the energy for executing the remaining part of the backup job. To achieve this goal, in [6] Mohammad *et. al* proposed to run the main tasks in the primary processor according to the earliest deadline as

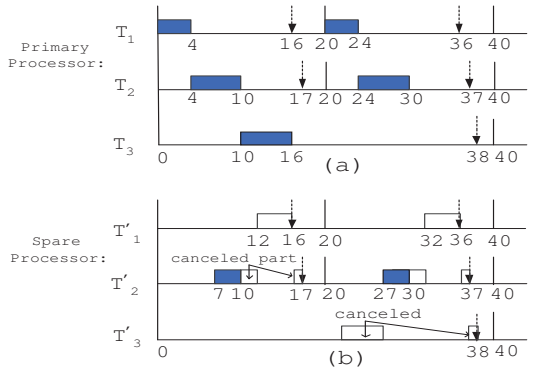

Fig. 1. (a) The schedule for the main tasks τ_1, τ_2, and τ_3 in the primary processor under the EDS scheme; (b) The schedule for the backup tasks $\tau_1^{'}$, $\tau_2^{'}$, and $\tau_3^{'}$ (which are canceled/partially-canceled based on [6]) in the spare processor under the EDL scheme [9].

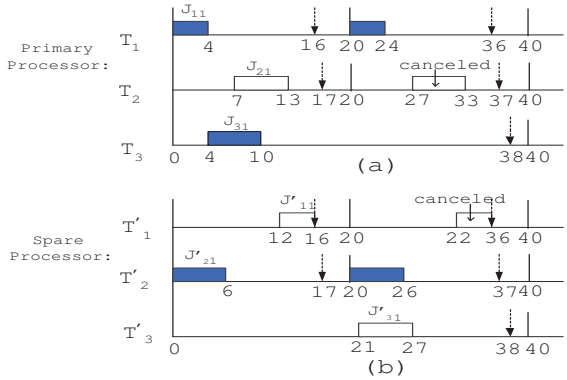

Fig. 2. The schedule for the main/backup jobs of each task under adaptive delay of individual job(s) in: (a) the primary processor; (b) the spare processor.

soon as possible (EDS) scheme and the backup tasks on the spare processor according to the earliest deadline as as late as possible (EDL) scheme [9] such that the overlapped executions between the main jobs and their backup jobs could be reduced, enabling energy savings. Their idea could be demonstrated using the following example:

Given a task set of three tasks, *i.e.*, $\tau_1 = (4,16,20,1,1)$, $\tau_2 = (6,17,20,1,1)$, $\tau_3 = (6,38,40,1,1)$, to be executed in a standby-sparing system. Since the QoS constraints for the tasks are all $(1,1)$-hard which equals to the periodic hard real-time case in which all jobs are "mandatory", we do not have optional jobs for these tasks. For simplicity, in this section all jobs refer to the mandatory jobs.

By applying the EDS-EDL scheme in [6], the main tasks τ_1 and τ_2, and τ_3 will be scheduled in the primary processor while backup tasks $\tau_1^{'}$ and $\tau_2^{'}$, and $\tau_3^{'}$ will be scheduled in the spare processor. If we assume no fault occurred, the complete schedules for them within the hyperperiod $[0,40]$ is shown in Figure 1(a) and (b), respectively. As a result, the total active energy consumption within the hyperperiod is 32 units.

Note that in the above example, there are still much overlapped time between the executions of all jobs of task τ_2 and their corresponding backup jobs, which costs significant energy consumption. On the other hand, as will be seen, if we adopt a different scheme of scheduling the task set, we can achieve better energy efficiency. Before presenting the new scheduling scheme in more details, we firstly introduce the following theorem for implementing the procrastinated execution of any job(s) in the task set (the proof is provided in the Appendix A part).

Theorem 1: Given a task set $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_N\}$, if the release time(s) of any job(s) under the EDS schedule is/are procrastinated to its/their corresponding delayed starting time(s)[1] under the EDL schedule, all task deadlines can be guaranteed.

To help understand Theorem 1, for the task set in Figure 1, it is easy to verify that if the release time(s) of any job(s) (for example, J_{21}) in Figure 1 (a) is/are procrastinated to the delayed release time(s) of the same job(s) in Figure 1 (b), (for example, $\hat{r}_{21} = 7$ for task J_{21}), all deadlines in Figure 1 (a) can be guaranteed.

With Theorem 1, our new approach for scheduling the task set under standby-sparing can be implemented based on the **adaptive delay of individual jobs**, which is shown in Figure 2. As seen in Figure 2, unlike the EDS-EDL scheme in Figure 1 which always (and only) delay all backup jobs in the spare processor uniformly, our approach will adaptively delay each individual job (either a main job or its corresponding backup job, but not both) depending on the actual need, which is summarized into the following *adaptive delay policies*:

- *Policy I*: at any time, if either a main job or its backup job, whichever first, gets chance to be dispatched and executed, it should be executed as soon as possible while the other one should be delayed to the starting time of the same job under EDL scheme and executed as late as possible.
- *Policy II*: whenever slack time becomes available, the undelayed job(s) should try to reclaim the slack time for execution (to facilitate early completion) while the delayed job(s) should never reclaim the slack time. Instead, it should try to utilize the slack time to be delayed further (under dynamic procrastination).

Based on the above adaptive delay policies, as seen in Figure 2(a), at time $t = 0$, both job J_{11} and its backup job $J_{11}^{'}$ got a chance to be dispatched. However, since only one of them can be executed as soon as possible and the other one must be delayed, we just randomly picked out one of them, say J_{11}, to be executed as soon as possible in the primary processor (Figure 2(a)) while delaying its backup job $J_{11}^{'}$ in the spare processor to time $\hat{r}_{11} = 12$ (Figure 2(b)). At the same time, since J_{21} will be preempted by J_{11} (due to its lower priority) and could not get chance to be dispatched/executed at time $t = 0$ in the primary processor (Figure 2(a)) while its backup job $J_{21}^{'}$ will get chance to be dispatched/executed at time $t = 0$ in the spare processor (Figure 2(b)), $J_{21}^{'}$ will be executed as soon as possible in the spare processor while J_{21} should be delayed to time $\hat{r}_{21} = 7$ in the primary processor. Similarly, at time $t = 4$, since J_{31} got chance to be dispatched/executed first, it will be executed as soon as possible while its backup job $J_{31}^{'}$ will be delayed to time $\hat{r}_{31} = 21$ and executed as late as possible. Following the same rationale, the complete schedules within the hyperperiod [0,40] are shown in Figure 2(a) and (b).

[1]Note that for the rest of the paper we use $\hat{r}_i$ to represent the delayed starting time of job J_i under the EDL schedule. Also when it does not cause any confusion, the delayed starting time has the same meaning as the delayed release time and they can be used exchangeably.

Under the same fault free assumption as in Figure 1, the total active energy consumption within the hyperperiod is reduced to 26 units, which is 19% lower than that in Figure 1.

It is not hard to see that in Figure 2 the fault tolerance capability of the standby-sparing system is preserved as whenever some job failed, its corresponding job in the other processor can still be executed and completed timely.

From the above example we can see that, by executing the tasks based on the adaptive delay policies above, there is great potential for energy saving. Based on the above principles, our standby-sparing scheduling algorithm for the $(1,1)$-hard tasks is presented in Algorithm 1.

Algorithm 1 The scheduling algorithm for $(1,1)$-hard tasks based on adaptive delay policies on individual job(s)

1: **For either the primary processor or the spare processor:**
2:
3: **Upon the execution of a job J_i at current time t_{cur}:**
4: **if** its category is "E" **then**
5: Execute it following the EDF scheme as soon as possible;
6: **if** any slack time $S_i(t)$ with higher priority than J_i is available **then**
7: Reclaim the slack time to execute J_i as soon as possible;
8: **end if**
9: **else**
10: // J_i's category is "D" and should be executed as late as possible;
11: Revise the arrival time of J_i to $\max\{\hat{r}_i,(t_{cur}+S_i(t_{cur}))\}$;
12: Execute J_i following the EDF scheme;
13: **end if**
14:
15: **Upon the completion of a job J_i at current time t_{cur}:**
16: **if** the execution of job J_i is successful **then**
17: Cancel its corresponding (backup) job in the other processor and add the residue time budget to the slack queue $\mathcal{S}$;
18: **if** J_i was the only job in the job ready queue at time t_{cur}^- **then**
19: Let NTA be the earliest (revised) arrival time of the next upcoming jobs(s) of all tasks;
20: **if** $(NTA-t_{cur})>T_{be}$ **then**
21: Shut down the processor and set wake-up timer as $(NTA-t_{cur})$;
22: **end if**
23: **end if**
24: **end if**

As shown in Algorithm 1, each job (either a main or backup job) J_i has a category field associated to it whose value could be "E" (representing as early as possible execution) or "D" (representing as late as possible delay). Upon dispatching, if J_i got chance to be executed earlier than its corresponding job in the other processor, J_i's category should be set as "E" which means it should always be executed as early as possible (for example, J_{31} in Figure 2(a)). Otherwise J_i's category should be set as "D" which means it should always be delayed as late as possible (for example, $J_{31}^{'}$ in Figure 2(b)). Whenever a job is completed successfully, its corresponding job in the other processor should be canceled and the remaining part of its time budget will become slack time (line 17).

Note that, during run-time, in both the primary and the spare processors, a slack queue $\mathcal{S}$ needs to be maintained to keep track of the slack time(s) from (partially) canceled job(s). The slack time(s) in $\mathcal{S}$ will be sorted according to their deadline(s). Upon job completion, new slack time from canceled job, if any, will be inserted into the slack queue $\mathcal{S}$ based on its deadline. Upon the dispatching of a job J_i at time t, the slack time from $\mathcal{S}$ with priorities higher than or equal to J_i will be stored in a variable $S_i(t)$. If J_i's category is "E", $S_i(t)$ should be reclaimed to execute J_i as soon as possible (line 6-8). Otherwise if J_i's category is "D", $S_i(t)$ should be used to implement dynamic procrastination of J_i, which can delay J_i to $\max\{\hat{r}_i,(t_{cur}+S_i(t))\}$ (line 11). But when the system is idle (or shut down), slack times in $\mathcal{S}$ should also be consumed based on their sorted sequence in $\mathcal{S}$.

The complexity of Algorithm 1 mainly comes from computing the delayed release time for the jobs based on EDL and the reclaimable slack time $S_i(t)$ for job J_i. Since the former can be computed offline and at anytime there are at most n jobs in the slack queue $\mathcal{S}$ of the primary processor or the spare processor, its online complexity is $O(n)$.

IV. Standby-sparing for general (m,k)-hard task sets

For tasks with general (m,k)-hard deadlines, to ensure the (m,k)-constraint, a widely adopted strategy is to judiciously partition the jobs into *mandatory* jobs and *optional* jobs [16]. Two well-known partitioning strategies are the *evenly distributed pattern* (or E-pattern) [14] and the *deeply-red pattern* (or R-pattern) [17]. According to E-pattern, the pattern π_{ij} for job J_{ij}, *i.e.*, the j^{th} job of a task τ_i, is defined by (here"1" represents the mandatory job and "0" represents the optional job):

$$\pi_{ij}=\begin{cases}\text{"1"} & \text{if } j=\lfloor\lceil\frac{(j-1)\times m_i}{k_i}\rceil\times\frac{k_i}{m_i}\rfloor\\ \text{"0"} & \text{otherwise}\end{cases}\qquad j=1,2,3,\cdots \tag{1}$$

And according to R-pattern, the pattern π_{ij} for job J_{ij} is defined by:

$$\pi_{ij}=\begin{cases}\text{"1"} & \text{if } 1\le j \bmod k_i\le m_i\\ \text{"0"} & \text{otherwise}\end{cases}\qquad j=1,2,3,\cdots \tag{2}$$

The mandatory/optional job partitioning according to equation (1) has the property that it helps to spread out the mandatory jobs *evenly* in each task along the time. Moreover, it is shown in [10] that E-pattern has better schedulability that R-pattern in general and is the optimal pattern when all task periods are co-prime in particular. In [10], a variation of E-pattern called E^R-pattern was achieved by reversing the pattern horizontally to let the optional jobs happen first, which can preserve the schedulability of E-pattern [10].

For task sets based on R-pattern, Niu *et al.* [18] proposed an approach by exploring the flexibility of executing jobs under (m,k)-deadlines to avoid executing duplicate copies of the mandatary jobs on two processors whenever possible. Their approach is based on selectively executing some optional jobs with flexibility degree of 1 (*i.e.*, the optional jobs right before the mandatory jobs) and, once they are completed successfully, their next mandatory job(s) will become optional

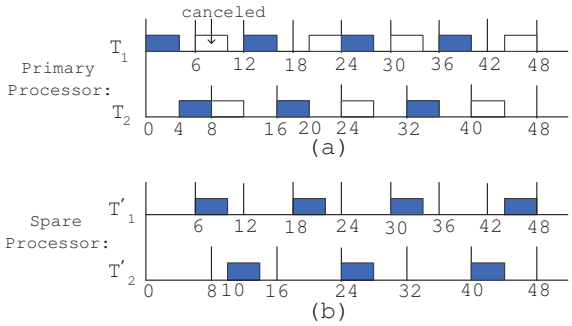

Fig. 3. (a) The schedule for the optional jobs (and canceled backup jobs) for task set $\tau_1 = (4,6,6,4,8)$, $\tau_2 = (4,8,8,2,4)$ in the primary processor under the selective scheme in [18] based on E-pattern; (b) The schedule for the optional jobs (and canceled backup jobs) for the same task set in the spare processor under the selective scheme in [18] based on E-pattern.

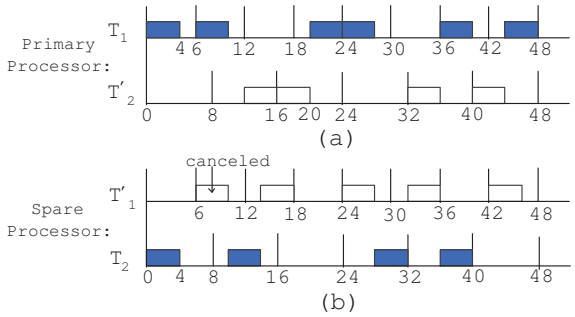

Fig. 4. The schedule for the mandatory main jobs based on E-pattern and (canceled) backup jobs based on E^R-pattern for same task set as in Figure 3 using our approach (a) in the primary processor; (b) in the spare processor.

(their backup jobs can be simply dropped to save energy) and this procedure can be progressed further. Their approach are generally efficient in saving energy for task sets already schedulable with R-pattern. However, as we mentioned, since the schedulability of R-pattern is worse than E(or E^R)-pattern, there still exist a number of task sets schedulable with E(or E^R)-pattern but not schedulable with R-pattern. If we apply the approach in [18] to the task sets partitioned based on E(or E^R)-pattern, it could execute excessive number of optional jobs, which could adversely affect the overall energy efficiency. This could be demonstrated with the following example.

Consider another task set of two tasks, *i.e.*, $\tau_1 = (4,6,6,4,8)$, $\tau_2 = (4,8,8,2,4)$. It is easy to verify that this task set is schedulable under E(or E^R)-pattern but not schedulable under R-pattern. The job patterns based on E^R-pattern for them are "01010101", and "0101", respectively. As can be seen, due to their even distribution property, all optional jobs in them have a flexibility degree of 1. If we apply the approach in [18] to the task set, the schedules in the primary and the spare processors are shown in Figure 3(a) and (b), respectively. As shown in Figure 3(a), since all optional jobs (including those jobs demoted from mandatory to optional) in it have a flexibility degree of 1, all of them will be selected for execution in the primary or the spare processor alternatively. If we assume no fault occurred within the first hyperperiod $[0,48]$, all mandatory main/backup jobs in either the primary or the spare processor could be demoted/dropped. As a result the total active energy consumption within the hyperperiod is 56 units.

However, if we follow a different way of scheduling the task set, we can achieve even better energy efficiency. Our new approach will be based on the following lemma to convert a given window-constraint into (m,k)-constraint automatically.

Lemma 1: [13] For any task τ_i with (m,k)-constraint of (m_i,k_i), if it can satisfy the *window-constraint* of $m_i/\frac{(m_i+k_i)}{2}$, its original (m,k)-constraint will be satisfied automatically.

The above lemma provides us more opportunities to reduce the energy consumption under standby-sparing. Specifically, we can determine the mandatory jobs of each main task τ_i and their backup jobs based on the window constraint of $m_i/\frac{(m_i+k_i)}{2}$ first (then according to Lemma 1, its original (m,k)-constraint will be satisfied automatically). For the tasks in the same task set as in Figure 3, their corresponding window constraints will be $4/6$ and $2/3$, respectively. Then under E-pattern, the mandatory jobs of task τ_1 will be scheduled in the primary processor, as shown in Figure 4(a) and the backup jobs of τ_1 will be determined based on E^R-pattern within each separate window of length $\frac{(m_i+k_i)}{2}$ and scheduled in the spare processor, as shown in Figure 4(b). Meanwhile, to balance the mandatory workload of two processors, we let the mandatory main jobs for task τ_2 based on E-pattern be scheduled in the spare processor and the backup jobs for them based on E^R-pattern be scheduled in the primary processor, as shown in Figure 4. As such, each mandatory main job and its backup job will be shifted away completely such that once a mandatory main job is completed successfully, its backup job in the other processor could be canceled completely. If any mandatory main job of task τ_i failed, its corresponding backup job in the other processor could still be invoked timely. In this way, the window-constraint of task τ_i could be guaranteed. Then according to Lemma 1, its original (m,k)-constraint can also be ensured. Following the same rationale, if we assume no fault occurred, the complete schedule within the hyperperiod is shown in Figure 4. The total active energy consumption for it is reduced to 40 units, which is 28.6% lower than that in Figure 3.

From the above example we can see that there is great potential for energy savings by determining the mandatory main/backup jobs based on window constraint first (and then convert to the original (m,k)-constraint). The main issues are: (i) when $\frac{(m_i+k_i)}{2}$ is not an integer, it is not a valid window length and thus cannot be used to implement window-(m,k)-constraint transferring in this way; (ii) since $\frac{(m_i+k_i)}{2} \leq k_i$, after the mandatory jobs are determined under window constraint based on Lemma 1, some task(s) might become non-schedulable. Therefore, it is possible that only part of the tasks, not all of them, could adopt the above method to save energy. Then the problem becomes how to determine the subsets of tasks to be partitioned with window-constraint and original (m,k)-constraint in a hybrid way to maximize the energy reduction. In this section, we first adopt a "branch-and-bound" method, similar to that in [19], to divide the task set $\mathcal{T}$ into two parts, *i.e.*, the subset $\mathcal{A}$ in which the tasks will be partitioned based on window-constraint and the subset $\mathcal{B}$ in which the tasks will be partitioned with original (m,k)-constraint. Before introducing our approach in detail, we first introduce the following theorem for checking the feasibility of

such kind of hybrid task sets consisting of subsets $\mathcal{A}$ and $\mathcal{B}$.

Theorem 2: Given system $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_N\}$. Let $\mathcal{B}$ contains the subset of tasks with mandatory jobs determined with the original (m,k)-constraints and $\mathcal{A}$ be the subsets of tasks with mandatory jobs determined through the corresponding window constraint $m_i/\frac{(m_i+k_i)}{2}$. Let L be the ending point of the first busy period for executing the mandatory jobs only and $LCM(T_i)$ be the *least common multiple* of $T_i, i = 1,2,...,N$. Then $\mathcal{T}$ is schedulable if for any mandatory job absolute deadline $d \leq \min\{L, LCM(T_i)\}$.

$$d \geq \sum_{\tau_a \in \mathcal{A}} \lceil \frac{m_a}{\frac{(m_a+k_a)}{2}} \lceil \frac{d-D_a}{T_a} \rceil^+ \rceil C_a + \sum_{\tau_b \in \mathcal{B}} \lceil \frac{m_b}{k_b} \lceil \frac{d-D_b}{T_b} \rceil^+ \rceil C_b \quad (3)$$

The right side of equation (3) represents the total work demand from the mandatory jobs in $\mathcal{A}$ and in $\mathcal{B}$ with absolute deadlines less than or equal to d. The proof for this theorem could be done in a similar way to that for Theorem 1 in [10] and is thus omitted.

Based on Theorem 2, our branch-and-bound approach is presented in Algorithm 2.

From Algorithm 2, our approach determines task by task if the mandatory main/backup jobs of each task should be determined based on the original (m,k)-constraint or based on the window-constraint first. When Algorithm 2 is finished, it is possible to reach certain hybrid configuration in which the tasks in $\mathcal{A}$ are partitioned based on window-constraints, while the tasks in $\mathcal{B}$ are still partitioned based on their original (m,k)-constraints. And the resulting configuration should be the one that could maximize the expected fault free energy reduction for the tasks in $\mathcal{A}$.

Algorithm 2 Determining the tasks adopting window-constraints.

Input: task set $\mathcal{T}$ with original (m,k)-constraint;
Output: task set $\mathcal{Z} = \mathcal{A} \cup \mathcal{B}$, where $\mathcal{A}$ is the subset of tasks in $\mathcal{T}$ adopting window-constraints and $\mathcal{B}$ is the subset of adopting original (m,k)-constraints;
$\mathcal{A} = \emptyset$; $\mathcal{B} = \mathcal{T}$; $\mathcal{Z} = \mathcal{A} \cup \mathcal{B}$; $E_{bd} = 0$
Try-Window-Constraint ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{Z}$, E_{bd});
Output ($\mathcal{Z}$);

FUNCTION Try-Window-Constraint ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{Z}$, E_{bd})
for each task $\tau_i \in \mathcal{B}$ **do**
  Re-determine the mandatory jobs of τ_i based on the *window-constraint* that can be converted to its original (m,k)-constraint;
  Remove τ_i from $\mathcal{B}$ and put it into $\mathcal{A}$;
  if $\mathcal{A} \cup \mathcal{B}$ is schedulable **then**
    $E_{save} = \sum_{\tau_i \in \mathcal{A}} \{2m_i \times \frac{LCM(k_i T_i)}{k_i} - m_i \times \frac{LCM(\frac{(m_i+k_i)}{2} T_i)}{\frac{(m_i+k_i)}{2}}\}$;
    //The expected fault free energy saving;
    if $E_{save} > E_{bd}$ **then**
      $E_{bd} = E_{save}$; $\mathcal{Z} = \mathcal{A} \cup \mathcal{B}$;
    end if
    Try-Window-Constraint ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{Z}$, E_{bd});
  else
    Restore the job patterns of τ_i to be based on its original (m,k)-constraint and put it back to $\mathcal{B}$;
  end if
end for

A. Execution of optional jobs

So far, we only considered energy saving for the tasks in $\mathcal{A}$. For the tasks in $\mathcal{B}$, we cannot adopt the same approach on them because if we use E-pattern to determine mandatory main jobs in the primary processor while using E^R-pattern to determine the backup jobs in the spare processor, it is possible that the (m,k)-constraint in some sliding window will be violated if some mandatory main job failed because E(or E^R)-pattern only contains the minimal number of mandatory jobs that "just" satisfy the (m,k)-constraint. In this case the failed mandatory main job could not be compensated by its backup job timely due to the strictness of the original (m,k) requirement. To save energy for the tasks in $\mathcal{B}$, a more promising way is to execute some optional job(s) when possible and vary the patterns of the future jobs correspondingly. This could be demonstrated using the following example.

Consider another task set of three tasks, *i.e.*, $\tau_1 = (2,2,5,2,3)$, $\tau_2 = (1,3,10,2,4)$, $\tau_3 = (3,6,15,1,3)$. After we applied the branch-and-bound method in Algorithm 2, only τ_2 and τ_3 can have their mandatory main/backup jobs determined based on the window-constraints of $2/3$ and $1/2$, respectively while τ_1 still needs to determine its mandatory main/backup jobs based on its original (m,k)-constraint due to schedulability. If we assume no fault occurred, the schedule within the first hyperperiod $[0,30]$ is shown in Figure 5 and the total active energy consumption for it is 21 units. Note that in this schedule no optional job was executed.

However, if we follow a different schedule which executes some optional job for τ_1, the energy efficiency could be improved further. As shown in 6(a), at time $t = 15$, the optional job J_{14} could be executed and completed timely. If the execution of J_{14} was successful, the next mandatory job J_{15} could be demoted to optional (its backup job in the spare processor could be dropped to save energy). Note that in order to ensure the original (m,k)-constraints, all future job patterns of task τ_1 could be varied by restarting the E^R-pattern from the next job position. Since the optional jobs do not need backup jobs for them, the execution of J_{14} should be quite helpful in reducing the energy for executing the mandatory main job J_{15} together with its backup job under the old patterns. Then at time $t = 20$, the demoted job J_{15} could be re-executed as an optional job and further demoted J_{16} to optional. Similarly, J_{16} could also be re-executed as an optional job at time $t = 25$. This procedure could be repeated dynamically and the complete schedule within the first hyperperiod is shown in Figure 6. The total active energy consumption for it is reduced to 19 units, which is 9.5% lower than that in Figure 5.

One critical issue in the above approach of executing the optional job(s) and vary the patterns dynamically is to ensure the original (m,k)-constraint be satisfied after pattern variation.

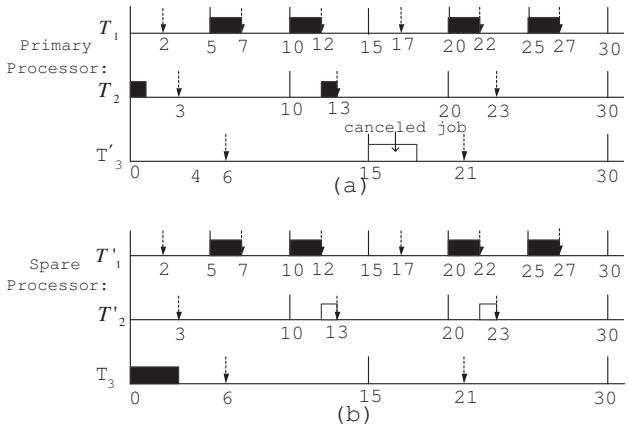

Fig. 5. (a) The schedule for the mandatory main jobs under static pattern; (b) The schedule for the backup jobs under static pattern.

Algorithm 3 The online algorithm for (m,k)-hard tasks

1: **For the primary processor:**
2: **if** MJQ is not empty **then**
3: Let J_i be the job with highest priority in MJQ.
4: **if** $\tau_i \in \mathcal{A}$ **then**
5: Execute J_i based on regular EDF scheme;
6: **else**
7: Execute J_i following the rationale in Algorithm 1;
8: **end if**
9: **if** J_i is completed successfully **then**
10: Let $J_i^{'}$ be the job of task $\tau_i^{'}$ in the other processor within the same time frame as J_i;
11: **if** $J_i^{'}$ is not the backup job of some other failed mandatory main job **then**
12: Cancel its corresponding job in the other processor and add the residue time budget to the slack queue $\mathcal{S}$;
13: **end if**
14: **end if**
15: **else if** OJQ is empty **then**
16: t_{cur} = the current time;
17: NTM = the earliest arrival of upcoming mandatory job(s);
18: **if** $(NTM - t_{cur}) > T_{be}$ **then**
19: Shut down the processor and set up its wake-up timer to be $(NTM - t_{cur})$;
20: **end if**
21: **else**
22: Select J_o in OJQ with the earliest deadline among jobs in OJQ with execution times no larger than $(\min\{NTM, d_o\} - t_{cur})$;
23: **if** $J_o \neq \emptyset$ **then**
24: Run J_o non-preemptively;
25: **if** J_o is completed successfully **then**
26: Restart the E^R-pattern of task τ_o from the next job position following J_o;
27: **end if**
28: **else**
29: Repeat lines 18-20;
30: **end if**
31: **end if**
32:
33: **For the spare processor:**
34: **if** $MJQ^{'}$ is not empty **then**
35: Run the jobs in $MJQ^{'}$ following the rationale in Algorithm 1;
36: **else**
37: Repeat lines 18-20;
38: **end if**

Fortunately, this could be guaranteed by Lemma 3 in [10].

B. The overall online algorithm

With the above information, our overall online algorithm for general (m,k)-hard tasks can be implemented in Algorithm 3.

As shown in Algorithm 3, in the primary processor, two job ready queues are maintained: the mandatory job queue (MJQ) and the optional job queue (OJQ). The jobs in MJQ always have higher priorities than those in OJQ. Upon arrival, a job of task τ_i is inserted into the MJQ if it is mandatory, regardless whether τ_i belongs to $\mathcal{A}$ or $\mathcal{B}$. On the other hand, an optional job of task τ_i is inserted into the OJQ only when τ_i belongs to $\mathcal{B}$ because only the tasks in $\mathcal{B}$ needs to run their optional jobs and vary their patterns when necessary. Moreover, to avoid executing excessive number of optional jobs, only the optional jobs right before the mandatory jobs should be inserted into the OJQ while all the other optional jobs should be skipped. In the spare processor, things will be different as, to avoid executing too many optional jobs (which could consume more energy than necessary), the optional jobs should not be executed there. As such, in the spare processor, only a mandatory backup job queue (represented as $MJQ^{'}$) will be maintained.

During runtime, for tasks in $\mathcal{A}$, only mandatory jobs will be executed based on regular EDF scheme. For tasks in $\mathcal{B}$, some optional jobs will be executed in the primary processor with dynamic pattern variation (lines 24-26). Note that if an optional job cannot be completed timely, it is not energy beneficial and therefore should not be invoked at all. As such, an optional job is regarded as eligible only if it could surely be finished before the earliest arrival time of the upcoming mandatory jobs (line 22). Moreover, once a selected optional job is invoked, it should be executed non-preemptively to ensure that it could be finished timely. Once an optional jobs is completed successfully, it will be counted as an effective job and the patterns of the future jobs should be varied by restarting the E^R-pattern of the same task in both the primary and the spare processors from the next job position. Note that if no optional jobs are available, the mandatory jobs in $\mathcal{B}$ can still be executed following the rationale in Algorithm 1 based on the adaptive delay policies in Section V-A (lines 7 and 35). The only issue is, in this case, since EDL scheme is not applicable for task sets with dynamic pattern variation, the delayed release time of any mandatory job J_i of task τ_i should be reset as $\hat{r}_i = r_i + Y_i$, where Y_i is defined as [20]

$$Y_i = D_i - R_i \tag{4}$$

where R_i be the worst case response time of task τ_i and can be computed off-line using the approach in [21].

Note that, for tasks in $\mathcal{A}$, when the current mandatory main job is completed successfully, whether its backup job in the other processor should be canceled or not needs to be handled carefully. Specifically, if it is within the same time frame of the backup job of some other failed mandatory job, its backup job cannot be canceled (line 11).

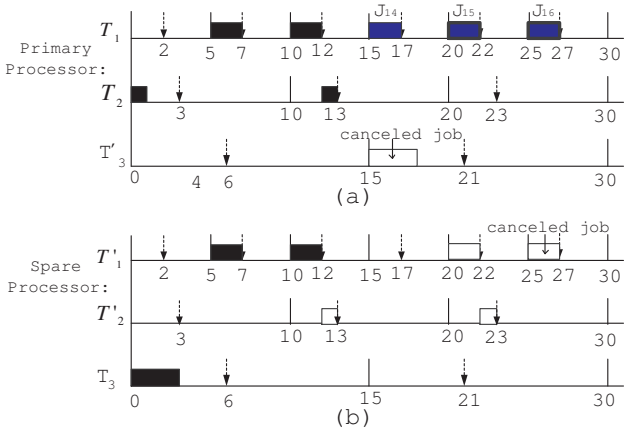

Fig. 6. (a) The schedule for the mandatory/optinal jobs after dynamic pattern shifting for task τ_1; (b) The schedule for the backup jobs after dynamic pattern variation for task τ_1.

The online complexity of Algorithm 3 mainly comes from scheduling the mandatory and optional jobs in the primary processor. Since at anytime there are at most n jobs in the MJQ or in the OJQ, its complexity is $O(n)$. Moreover, to ensure that the (m,k)-constraint of the tasks be satisfied, we have the following theorem (proof omitted due to page limit):

Theorem 3: Let task set $\mathcal{T}$ be scheduled with Algorithm 3. The (m,k)-deadlines for $\mathcal{T}$ can be ensured if $\mathcal{T}$ is schedulable under E-pattern.

V. Evaluation

In this section, we compare the energy performance of our approach with other previous approaches using simulations. We conducted two groups of simulations, one for $(1,1)$-hard task sets and one for general (m,k)-hard task sets.

A. Simulation results for $(1,1)$-hard task sets

Three different approaches are studied. The first approach (NEM_{SS}) executes the jobs in the primary and the spare processors concurrently without energy management. We use its results as the reference. The second approach (EM_{SS}) schedule the tasks with the EDS-EDL scheme from [6] without applying DVFS. The third approach ($ADIJ_{SS}$) is our approach proposed in Section III based on adaptive delay policies on individual jobs in both the primary and the spare processors. We assume the processor shut-down break even time $T_{be} = 1ms$.

The periodic task set in our experiments consists of five to ten tasks with the periods randomly chosen in the range of [5, 50]ms and the deadlines were assumed to be less than or equal to their periods. The worst case execution time (WCET) of a task was assumed to be uniformly distributed and the total utilization, *i.e.*, $\sum_i \frac{C_i}{P_i}$ was divided into intervals of length 0.1 each of which contains at least 20 task sets schedulable or at least 5000 task sets generated. We conducted two sets of tests.

In the first set, we check the energy performance when no fault occurred within the hyperperiod. The result is shown in Figure 7(a).

From Figure 7(a), one can immediately see that, by adopting the adaptive delay policies on each main/backup job individually, $ADIJ_{SS}$ can achieve much better energy efficiency

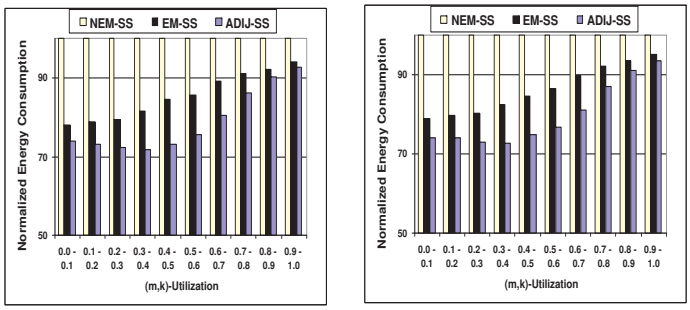

Fig. 7. The results subject to (a) No faults; (b) System faults.

than the previous approaches, *i.e.*, NEM_{SS} and EM_{SS}, in most utilization intervals. The energy reduction by $ADIJ_{SS}$ over EM_{SS} can be up to 14%. The main reason is that, in this scenario, by executing the main/backup jobs in a more flexible way, $ADIJ_{SS}$ can help reduce the overlapped execution between the main jobs and their backup jobs more efficiently, therefore saving more energy.

In the second set, we assumed the system could be subject to permanent and/or transient faults. The transient fault model is similar to that in [1] by assuming Poisson distribution with an average fault rate of 10^{-5}. The result is shown in Figure 7(b).

As seen, in this scenario, the energy saving by our new approaches, *i.e.*, $ADIJ_{SS}$ is similar to that under the fault-free assumption. The energy reduction by $ADIJ_{SS}$ over EM_{SS} can be up to 12%, thanks to the adaptive executions of the jobs under individual/flexible delay.

B. Simulation results for general (m,k)-hard task sets

In this part, we also studied three approaches. The first approach (NEM_{MKE}) statically determine the job patterns based on E-pattern. And the mandatory jobs in the primary and the spare processors are executed concurrently without delay. We used its results as the reference. The second approach ($MKSS_{Selective}$) also determines the job patters based on E-pattern first but selectively executes the optional jobs using the approach in [18]. The third approach ($WCMK_{SS}$) is our new approach proposed in Section IV.

The periodic task sets are generated in the same way as in Section V-A but with m_i and k_i values for the (m,k)-constraint randomly generated between 2 and 10 ($k_i > m_i$). Since when the total (m,k)-utilization, *i.e.*, $\sum_i \frac{m_i C_i}{k_i P_i}$ is larger than 0.8, it is hard for the task sets to be schedulable, we mainly checked the task sets with (m,k)-utilization between 0.0 to 0.8. We also conducted two sets of tests.

In the first set, we checked the energy performance when no fault occurred within the hyperperiod. The result is shown in Figure 8(a).

From Figure 8(a), it is easy to see that $WCMK_{SS}$ can achieve much better energy performance than the previous approaches, *i.e.*, NEM_{MKE} and $MKSS_{Selective}$. The energy reduction by $WCMK_{SS}$ over $MKSS_{Selective}$ can be up to 20%. The main reason is that, in this scenario, by partitioning the jobs based on window-constraints (that could be converted to the original (m,k)-constraints) first, $WCMK_{SS}$ can help minimize

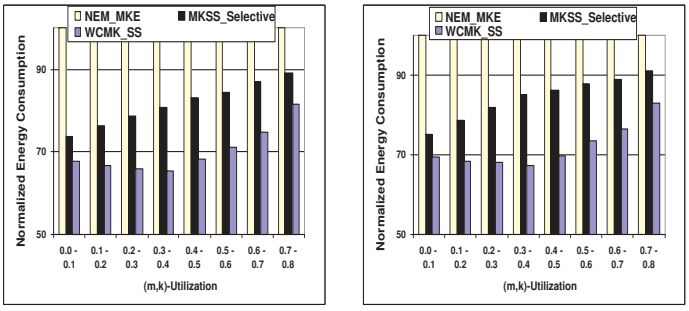

Fig. 8. The results subject to (a) No faults; (b) System faults.

the overlapped execution between the mandatory and backup jobs in two processors more efficiently. Moreover, for those tasks that cannot be applied window-constraint, $WCMK_{SS}$, with more adaptive optional job execution strategy, can avoid executing excessive number of optional jobs. In addition, when no optional jobs are available, by letting the mandatory main/backup jobs be delayed following the adaptive delay policies on individual job(s) in Section V-A, $WCMK_{SS}$ can also help save more energy for running the mandatory main/backup jobs when necessary.

In the second set, we assumed the system could be subject to permanent and/or transient faults with same fault rate as in Section V-A. The result is shown in Figure 8(b).

As seen, under this scenario, the energy saving achievable by our new approach, *i.e.*, $WCMK_{SS}$ over the previous approach is even better. The energy reduction by $WCMK_{SS}$ over $MKSS_{Selective}$ can be up to 22%. This is mainly due to the fact that the energy efficiency of $MKSS_{Selective}$ is highly dependant on the successfully execution of optional jobs and the dynamic pattern variation based on it. As such, when the system fault(s) occurred, the dynamic pattern variation procedure in $MKSS_{Selective}$ could be affected significantly. Different from that, in our new approach , *i.e.*, $WCMK_{SS}$ the execution of the optional jobs under dynamic pattern shifting only partially contributed to the overall energy reduction. A more important part of the energy saving in $WCMK_{SS}$ comes from our more flexible job partitioning strategy based on window-constraint (that could be transferred to the original (m,k)-constraint) as well as the adaptive executions of the mandatory main/backup jobs based on flexible delay when necessary.

VI. Conclusion

Energy consumption, fault-tolerance, and quality of service are becoming increasingly critical factors in the design of pervasive computing systems. In this paper, we presented two novel approaches to reduce the energy consumption for weakly hard real-time embedded systems under standby-sparing: one for $(1,1)$-hard tasks and one for general (m,k)-hard tasks. Through extensive evaluations, our results demonstrate that the proposed techniques significantly outperform the previous research in reducing energy consumption for both $(1,1)$-hard task sets and general (m,k)-hard task sets while assuring fault tolerance through standby-sparing.

Acknowledge*

This work is supported in part by NSF under project HRD-1800403.

Appendix A
Proof of Theorem 1

Proof: We use contradiction. Assuming under the EDS schedule, after the release time(s) of some job(s) are delayed to their starting times (*i.e.*, the delayed release times) under the EDL scheule, at certain time point t', some task missed its deadline. Then we can always find another time point $t_0 < t'$ such that during the time interval $[t_0,t']$ the processor is kept busy executing only jobs with release times or delayed release times no earlier than t_0 and with deadlines less than or equal to $(t'-t_0)$. Since no job has release time earlier than time 0, t_0 is well defined. Then the total work demand within the interval $[t_0,t']$ is bounded by $\sum_{D_q\leq(t'-t_0)}\lceil\frac{t'-t_0-D_q}{T_q}\rceil^+C_q$. Since some job missed the deadline at t', we have

$$\sum_{D_q\leq(t'-t_0)}\lceil\frac{t'-t_0-D_q}{T_q}\rceil^+C_q > (t'-t_0) \tag{5}$$

On the other hand, consider the scenario when we delay the release times of all jobs within the interval $[t_0,t']$ to their starting times under the EDL schedule. Then in this case there must be a time point t_1 $(t_0\leq t_1<t')$ such that during the interval $[t_0,t_1]$ the processor is either idle or executing jobs with deadlines larger than $(t'-t_0)$ while during the interval $[t_1,t']$ the processor is busy executing only the jobs with deadlines less than or equal to $(t'-t_0)$. Moreover, the total work demand within $[t_1,t']$ will be no larger than the total work demand within the interval $[t_1,t']$ under the EDL schedule, *i.e.*,

$$\sum_{D_q\leq(t'-t_1)}\lceil\frac{t'-t_1-D_q}{T_q}\rceil^+C_q \leq \sum_{D_q\leq(t'-t_1)}^{EDL}\lceil\frac{t'-t_1-D_q}{T_q}\rceil^+C_q \tag{6}$$

Since there is no deadline missing under the EDL schedule, we have

$$\sum_{D_q\leq(t'-t_1)}^{EDL}\lceil\frac{t'-t_1-D_q}{T_q}\rceil^+C_q \leq (t'-t_1) \tag{7}$$

Therefore, we have

$$\sum_{D_q\leq(t'-t_1)}\lceil\frac{t'-t_1-D_q}{T_q}\rceil^+C_q \leq (t'-t_1) \tag{8}$$

Meanwhile, since after delay the processor is idle or executing jobs with deadlines larger than $(t'-t_0)$ between $[t_0,t_1]$, the work demand within $[t_1,t']$ is the same as the work demand within $[t_0,t']$. Thus we have

$$\sum_{D_q\leq(t'-t_1)}\lceil\frac{t'-t_1-D_q}{T_q}\rceil^+C_q = \sum_{D_q\leq(t'-t_0)}\lceil\frac{t'-t_0-D_q}{T_q}\rceil^+C_q \tag{9}$$

Also since $t_0\leq t_1<t'$, we have $(t'-t_0)\geq(t'-t_1)$. Therefore we have

$$\sum_{D_q\leq(t'-t_0)}\lceil\frac{t'-t_1-D_q}{T_q}\rceil^+C_q \leq (t'-t_1) \leq (t'-t_0) \tag{10}$$

Which Contradicts to (5)! □

REFERENCES

[1] D. Zhu, R. Melhem, and D. Mosse, "The effects of energy management on reliability in real-time embedded systems," in *ICCAD*, 2004.

[2] J. S. A. S. B. R.J. and C.-K. Hu, "Ramp: A model for reliability aware microprocessor design," *IBM Research Report, RC23048*, 2003.

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

[4] Y. wen Zhang, H. zhen Zhang, and C. Wang, "Reliability-aware low energy scheduling in real time systems with shared resources," *Microprocessors and Microsystems*, vol. 52, pp. 312 – 324, 2017.

[5] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "Low-energy standby-sparing for hard real-time systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 329–342, March 2012.

[6] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware standby-sparing technique for periodic real-time applications," in *ICCD*, 2011.

[7] Y. Guo, H. Su, D. Zhu, and H. Aydin, "Preference-oriented real-time scheduling and its application in fault-tolerant systems," *Journal of Systems Architecture*, vol. 61, 01 2015.

[8] Y. wen Zhang, "Energy-aware mixed partitioning scheduling in standby-sparing systems," *Computer Standards and Interfaces*, vol. 61, pp. 129 – 136, 2019.

[9] H. Chetto and M. Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Transction On Software Engineering*, vol. 15, 1989.

[10] 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*, pp. 717–729, July 2006.

[11] ——, "Peripheral-conscious energy-efficient scheduling for weakly hard real-time systems," *International Journal of Embedded Systems*, vol. 7, no. 1, pp. 11–25, 2015.

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

[13] R. West, Y. Zhang, K. Schwan, and C. Poellabauer, "Dynamic window-constrained scheduling of real-time streams in media servers," *IEEE Trans. on Computers*, vol. 53, no. 6, pp. 744–759, June 2004.

[14] P. Ramanathan, "Overload management in real-time control applications using (m,k)-firm guarantee," *IEEE Trans. on Paral. and Dist. Sys.*, vol. 10, no. 6, pp. 549–559, Jun 1999.

[15] D. K. Pradhan, Ed., *Fault-tolerant Computing: Theory and Techniques; Vol. 2*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[16] G. Quan and X. Hu, "Enhanced fixed-priority scheduling with (m,k)-firm guarantee," in *RTSS*, 2000, pp. 79–88.

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

[18] L. Niu and D. Zhu, "Reliable and energy-aware fixed-priority (m,k)-deadlines enforcement with standby-sparing," *DATE*, 2020.

[19] L. Niu and J. Xu, "Improving schedulability and energy efficiency for window-constrained real-time systems with reliability requirement," *Journal of Systems Architecture*, vol. 61, no. 5-6, pp. 210–226, May-June 2015.

[20] L. Niu and D. Zhu, "Reliability-aware scheduling for reducing system-wide energy consumption for weakly hard real-time systems," *Journal of Systems Architecture*, vol. 78, pp. 30 – 54, 2017.

[21] J. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo, *Deadline Scheduling for Real-Time Systems EDF and Related Algorithms*. Berlin, Germany: Springer, 1998.