Energy-Constrained Standby-Sparing for Weakly Hard Real-Time Systems

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

***Abstract*—For real-time embedded systems, QoS (Quality of Service), fault tolerance, and energy budget constraint are among the primary design concerns. In this research, we investigate the problem of energy constrained standby-sparing for weakly hard real-time systems. The standby-sparing systems adopt a primary processor and a spare processor to provide fault tolerance for both permanent and transient faults. With the purpose of ensuring the feasibility of such kind of systems, we firstly present two novel scheduling schemes for the standby-sparing systems with tighter energy budget constraint than the traditional ones. Then based on them a hybrid approach is proposed to achieve better performance. The evaluation results demonstrated that the proposed techniques significantly outperformed the existing state of the art approaches in terms of feasibility and system performance while ensuring QoS and fault tolerance under given energy budget constraint.**

***Index Terms*—energy constraint, fault tolerance, QoS**

I. Introduction

With the advance of IC technology, energy constraint has been an increasingly important factor for the design of real-time embedded systems. In some real-time applications, the systems are driven by power supplies with limited energy budget constraint, which has to remain operational during a well-defined mission cycle. Examples include Heart Pacemakers [1] or other portable embedded devices whose power supply can only be charged to full capacity right before the beginning of certain mission/operation cycle/period(s). For such kind of applications, efforts must be made by all means to avoid exhausting the energy budget before the end of the mission cycle. On the other hand, fault tolerance has also been a major concern for pervasive computing systems as system fault(s) could occur anytime [2]. Generally, computing system faults can be classified into permanent faults and transient faults [3]. 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. [4], [5]) have been conducted on dealing with energy consumption 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 catastrophical 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 [6]–[9]. 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 [8]. 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, due to the deadline constraint, 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 in time. Thus the total energy consumption could be quite considerable. Regarding that, some recent works (e.g. [6]–[9]) have been reported to reduce energy by letting the executions of the main jobs and their backup jobs be shifted away such that, once the main jobs are completed successfully, their corresponding backup jobs could be canceled early. For standby-sparing systems with mixed criticality, advanced energy management schemes were proposed in [10]. When considering the chip thermal effect, peak-power-aware standby-sparing techniques utilizing energy management schemes were presented in [11].

All of the above 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 critical data could be lost.

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) [12]. Two well known weakly hard QoS models are the (m,k)-model [13] and the *window-constrained* model [14]. 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.* [15] 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 satisfied.

With energy budget constraint in mind, in [16], Zhao *et al.* proposed an approach to maximize the overall reliability of the systems under given time and energy constraints. Their approach only considered the transient faults. When both permanent and transient faults are taken into consideration in the context of standby-sparing, the energy-constrained issue is especially critical as the backup jobs in the spare processor could also incur significant energy consumption which could make the total energy consumption beyond the given energy budget constraint. In this paper, we study the problem of energy constrained standby-sparing for weakly hard real-time systems under the requirement of tolerating both permanent and transient faults. To the best of our knowledge, this is the first work to explore improving feasibility and performance of standby-sparing systems under given energy budget constraint.

The rest of the paper is organized as follows. Section II presents the preliminaries. Section III presents our approach based on the floating redundant job scheme. Section IV presents our approach based on the window transferring scheme. Section V presents our hybrid approach. In Section VI and Section VII, we present our evaluation results and conclusions.

II. Preliminaries

A. System model

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.*, (D_i, P_i, C_i, m_i, k_i). D_i ($\leq P_i$), P_i, and C_i represent the deadline, period, and worst case execution time (WCET) for τ_i respectively. A pair of integers, *i.e.*, (m_i,k_i) $(0 < m_i \leq k_i)$, are used to denote the (m,k)-constraint for task τ_i which requires that, among any k_i consecutive jobs, at least m_i jobs must be 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 denote its release time, execution time, and absolute deadline, respectively.

We assume the task set is to be executed in a standby-sparing system with a limited energy budget/supply of $\bar{E}_c$ units during its mission cycle. Moreover, we assume this energy budget is a hard constraint in a sense that it cannot be exceeded at any time during its mission cycle. Without loss of generality, we let the mission cycle be the hyper period of the task set, *i.e.*, $H = LCM(k_iP_i)$ and assume that the energy supply can only be charged to full capacity right before the beginning of each mission cycle.

The standby-sparing 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 *main task* and its corresponding copy running in the other processor *backup task*, denoted as $\tau_i^{'}$. The j^{th} job of task $\tau_i^{'}$ is denoted as $J_{ij}^{'}$ Moreover, we call each mandatory job J_{ij} of task τ_i *main job* and its corresponding job running in the other processor (to compensate its failure, if happened) *backup job*, denoted as $\tilde{J}_{ij}$. Note that in this paper J_{ij}'s backup job, *i.e.*, $\tilde{J}_{ij}$ might be different from $J_{ij}^{'}$, *i.e.*, the job of $\tau_i^{'}$ in the same time frame as J_{ij} because, as will be shown in later part of this paper, J_{ij} and $\tilde{J}_{ij}$ can be shifted away from each other completely such that they might belong to different time frames.

B. Energy Model

The processor can be in one of the three states: *busy*, *idle* and *sleeping* states. When the processor is busy executing a job, it consumes the busy power (denoted as P_{busy}) which includes dynamic and static components during its active operation. The dynamic power (P_{dyn}) consists of the switching power for charging and discharging the load capacitance, and the short circuit power due to the non-zero rising and falling

time of the input and output signals. The dynamic power can be represented [17] as

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

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

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

where I_{st} is mainly due to the leakage current which consists of both the subthreshold leakage current and the reverse bias junction current in the CMOS circuit. The power consumption when the processor is busy, *i.e*, P_{busy}, is thus

$$P_{busy} = P_{dyn} + P_{st}, \tag{3}$$

When the processor is idle, it consumes the idle power (denoted as P_{idle}) whose major portion comes from the static power. When the processor is in the sleeping state, it consumes the sleeping power (denoted as P_{sleep}) which is assumed to be negligible. Note that although dynamic power can be reduced effectively by dynamic voltage scaling (DVS) techniques, the efficiency of DVS 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), *i.e.*, put the processor into its sleeping state, on the other hand, can greatly reduce the leakage energy when the processor is not in use. With that in mind, in this paper we assume that, when the processors is busy, it always consumes P_{busy} at the maximal supply voltage V_{max}. Without loss of generality, we normalize P_{busy} and the processor speed under V_{max} (denoted as s_{max}) 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 sleeping state with DPD. Assume that the energy overhead and the timing overhead of shutting-down/waking-up the processor are E_o and t_o, respectively. Then the processor can be shut down with positive energy gains only when the length of the idle interval is larger than $t_{sd} = \max(\frac{E_o}{P_{idle} - P_{sleep}}, t_o)$. We therefore call t_{sd} the *minimal shut-down interval*.

C. Fault Model

Similar to the standby-sparing systems in [7], [8], 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 [18] 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.

D. Problem Formulation

Based on the above system models, the problem to be solved in this paper can be formulated as followed:

Problem 1: Given system $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_N\}$, schedule $\mathcal{T}$ with EDF scheme in a standby-sparing system such that the total energy consumption does not exceed the given energy budget constraint $\bar{E}_c$ while satisfying the (m,k)-constraints for all tasks under the fault tolerant requirement.

III. Energy-constrained standby-sparing based on floating redundant job scheme

To solve Problem 1, one essential part is to determine the *mandatory* jobs to be scheduled under standby-sparing. A widely known strategy to do so is to adopt the *evenly distributed pattern* (or E-pattern) [15] to partition the jobs into mandatory jobs and optional jobs. 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:

$$\pi_{ij} = \begin{cases} \text{“1”} & \text{if } j = \lfloor \lceil \frac{(j-1)\times m_i}{k_i} \rceil \times \frac{k_i}{m_i} \rfloor + 1 \\ \text{“0”} & \text{otherwise} \end{cases} \qquad j = 1, 2, \cdots \tag{4}$$

where “1” represents the mandatory job and “0” represents the optional job.

Note that the job patterns defined with E-pattern have the property that they define a *minimal set* of mandatory jobs that “just” satisfies the given (m,k)-constraint in each sliding window. Due to this property, in order to ensure the system reliability under standby-sparing, a popular approach is to reserve a backup job in the same time frame of the backup task running in the other processor for each mandatory job of the main task. Consequently, the total energy consumption will be two times of that consumed by one processor, *i.e.*

$$E = 2(\sum_i \frac{H m_i}{k_i} C_i + P_{idle} H (1 - \sum_i \frac{m_i C_i}{k_i P_i})) \tag{5}$$

where H is the hyper period.

From Equation (5), the energy consumed in a standby-sparing system could be quite considerable. In order to reduce energy consumption, in [7], Haque *et. al* proposed to run the main tasks/jobs in the primary processor according to the earliest deadline as soon as possible (EDS) scheme while the backup tasks/jobs in the spare processor according to the earliest deadline as late as possible (EDL) scheme [19] such that, once the main tasks/jobs are completed successfully, their corresponding backup tasks/jobs could be (partially) canceled. In [8], a more advanced technique named preference oriented scheme was adopted which, in both the primary and backup processors, lets some tasks be scheduled under EDS scheme while the other tasks be scheduled under EDL scheme. In [20], an energy-aware approach based on the execution of optional jobs was proposed for task sets partitioned under deeply-red pattern [21] which is weaker than E-pattern in ensuring the schedulability of the task sets [22]. Although the approaches in [7], [8], [20] are able to reduce the actual energy consumption of the standby-sparing system to some extent, since none of

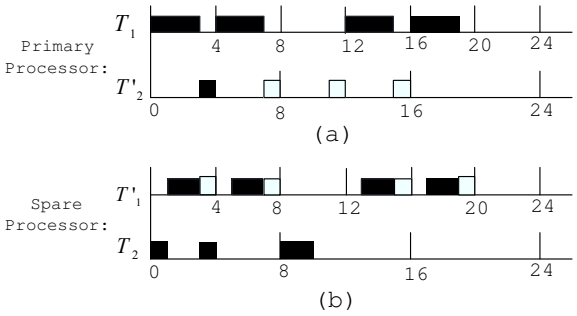


Fig. 1. The schedule for the mandatory main/bakcup jobs under the preference oriented scheme in [8]: (a) in the primary processor; (b) in the spare processor.

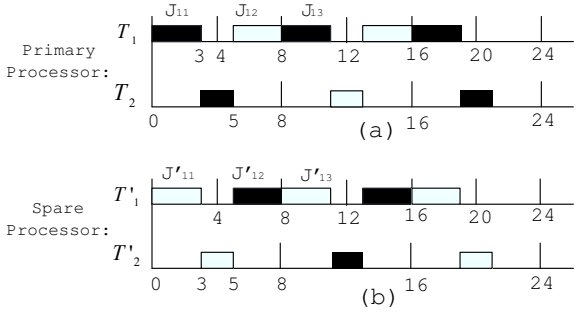


Fig. 2. The schedule for the mandatory main/bakcup jobs under the floating redundant job scheme: (a) in primary processor; (b) in spare processor.

them could predict the quantifiable amount of energy that can be saved in advance, the total energy budget still has to be at least the energy computed in Equation (5) in order to ensure the systems feasibility for the worst case. Otherwise if the given energy budget constraint $\bar{E}_c$ during the hyper period is less than the energy consumption computed with Equation (5), the task set can not be guaranteed to be feasible in advance. Regrading that, some more advanced technique needs to be explored in order to ensure the feasibility of the task set under tighter given energy budget constraint $\bar{E}_c$. This could be illustrated using the following example.

Consider a task set of two tasks, *i.e.*, $\tau_1 = (4,4,3,4,6)$, and $\tau_2 = (8,8,2,2,3)$, to be executed in a standby-sparing system with given energy budget constraint $\bar{E}_c = 28$ units within its hyper period 24.

If we assume no fault occurred during the hyper period, Figure 1 shows the schedule for the mandatory jobs based on E-pattern for the original given (m,k)-constraints based on the preference oriented scheme in [8] (the empty rectangles represent the canceled part of the jobs). Note that although in the result schedule the total energy consumption could be reduced by 7 units, this amount of energy reduction cannot be accurately estimated in advance, especially considering the possible transient/permanent faults that could happen anytime during the job execution. Therefore, in order to prepare for the worst case, we still need to assume the total busy energy consumption to be twice of that for executing the mandatory jobs in one processor, which is 32 units and has already exceeded the given energy budget constraint. As a result, the feasibility of the task set cannot be ensured.

However, if we adopt a different way of scheduling the task set, it is still possible to ensure the feasibility of the system. The main idea is: we firstly temporarily increase the m_i values of each task τ_i by 1 such that the (m,k)-constraints of tasks τ_1 and τ_2 become $(5,6)$ and $(3,3)$, respectively; after that for each task we use one of its mandatory jobs under the new (m,k)-constraint as the "temporary extra mandatory job" to help us reduce the energy budget required. The detailed schedule are shown in Figure 2. As shown Figure 2(a), for task τ_1, since its new job pattern under the new temporary (m,k)-constraint is "111110" which contains an extra mandatory job in it, this extra mandatory job does not need to have a backup job for it (because even if it had failed, the remaining ones can still satisfy the original (m,k)-constraint). As shown Figure 2, in the beginning we designated the first mandatory job of τ_1, *i.e.*, J_{11} in the primary processor as the temporary extra mandatory job and executed it without backup job at all (its backup job J'_{11} was canceled as soon as J_{11} was designated as the temporary extra mandatory job). Once J_{11} was completed successfully at time 3, we switched the temporary extra mandatory job to J'_{12} in the spare processor while canceling J_{12}. After J'_{12} was completed at time 8, we switched the temporary extra mandatory job to J_{13} in the primary processor while canceling J'_{13} ... This procedure could be repeated until all mandatory jobs of τ_1 under its new temporary (m,k)-constraint had been executed. The procedure for task τ_2 could also be conducted in a similar way. From Figure 2 it is not hard to see that, if no fault occurred during the hyper period, each task τ_i will have totally (m_i+1) mandatory jobs executed in either the primary or the spare processor within each window of k_i jobs. Therefore the total busy energy consumption within the hyper period will be 21 units. Even when we have the energy consumption during the idle period included, assuming the idle power of the processor P_{idle} to be 0.05, the estimated total energy consumption of the system within the hyper period will be (21 + $P_{idle} \times 27$=) 22.35 units, which is less than $\bar{E}_c$ and therefore feasible.

The above calculation is based on the assumption that no fault ever occurred. If during runtime a permanent fault occurred to one processor, only the mandatory jobs in the other processor will be executed to resume the system, which will not increase the total energy consumption computed above. On the other hand, if during runtime some transient fault(s) occurred, some temporary extra mandatory job might be failed due to it. In this case all the other mandatory jobs within the same window of k_i jobs become required ones whose backup jobs also need to be executed. Under this scenario the estimation of the total energy consumption also needs to take the energy consumption of those backup jobs into consideration based on probability. For example, in the above task set, if we assume the probability of transient fault to be 10^{-5}, then the expected energy consumption of all backup jobs within one window of k_i jobs will be $(3\times 4+2\times 2)\times 10^{-5} = 0.00016$ units. After adding it to the above result, the total estimated energy consumption of the system subject to fault(s) will be 22.35016 units, which is still less than the given energy budget constraint and therefore feasible.

Note that in the above approach the mandatory main/backup

Algorithm 1 The algorithm based on floating redundant job

1: **Preparations:** For each task $\tau_i \in \mathcal{T}$, re-partition it based on its new temporary (m,k)-constraint of (m_i+1,k_i) and determine its mandatory main/backup jobs in both primary and spare processors correspondingly. In primary processor, mark J_{i1}, *i.e.*, the first job of each task τ_i as its initial floating redundant job;
2:
3: **For either the primary processor or the spare processor:**
4:
5: **Upon the execution of a mandatory job J_{ij} at time t_{cur}:**
6: **if** J_{ij} is the floating redundant job **then**
7: Cancel J_{ij}'s corresponding job in the other processor and add its time budget to the slack queue $\mathcal{STQ}$;
8: Execute J_{ij} following the EDF scheme;
9: **if** any slack time $STQ_i(t)$ with higher priority than J_{ij} is available **then**
10: Reclaim the slack time to execute J_{ij} as soon as possible;
11: **end if**
12: **else if** J_{ij} is within the same window of k_i jobs as the most recent failed floating redundant job **then**
13: **if** J_{ij} is a mandatory main job **then**
14: repeat lines 8-11;
15: **else**
16: Revise r_{ij} to $\max\{(r_{ij}+\varphi_i),(t_{cur}+STQ_i(t_{cur}))\}$;
17: Execute J_{ij} following the EDF scheme;
18: **end if**
19: **else**
20: mark J_{ij} as the current floating redundant job;
21: **end if**
22:
23: **Upon the completion of a job J_{ij} at current time t_{cur}:**
24: **if** the execution of job J_{ij} is successful **then**
25: **if** J_{ij} is the floating redundant job **then**
26: Let J_a be the next mandatory job after J_{ij} in the other processor;
27: Mark J_a as the floating redundant job;
28: Cancel J_a's corresponding job in the other processor and add its time budget to the slack queue $\mathcal{S}$;
29: **else**
30: Cancel J_{ij}'s corresponding job in the other processor and add its residue time budget to the slack queue $\mathcal{S}$;
31: **end if**
32: **if** J_{ij} was the only job in the mandatory job queue at time t_{cur} **then**
33: Let NTA be the earliest arrival time of the next upcoming mandatory job in the same processor;
34: **if** $(NTA-t_{cur})>t_{sd}$ **then**
35: Shut down the processor and set wake-up timer as $(NTA-t_{cur})$;
36: **end if**
37: **end if**
38: **end if**

jobs of each task under the new temporary (m,k)-constraint was used as the temporary extra mandatory job alternatively. It appears in effect as if the temporary extra mandatory job was "floating" through the mandatory main/backup jobs one by one within each window of k_i jobs and jumping back and forth between the primary and the spare processors. Since this temporary extra mandatory job is not required for satisfying the original (m,k)-constraint, for convenience, we call it *floating redundant job*. As shown, this floating redundant job is very useful in helping us to reduce the estimation of the total energy consumption and meeting the overall energy budget constraint. Correspondingly the above approach is also called the *floating redundant job scheme*. The details of it are presented in Algorithm 1.

As shown in Algorithm 1, in the beginning, for each task $\tau_i \in \mathcal{T}$, we firstly re-partition it with its new temporary (m,k)-constraint of (m_i+1,k_i) based on E-pattern and mark its first job (represented as J_{i1}) as its initial floating redundant job (note that each task has a floating redundant job of its own). During runtime, in both the primary and the spare processors, a mandatory job ready queue (MQ) is maintained. Upon arrival, a job of task τ_i is inserted into the MQ if its job pattern is "1". All jobs in MQ will be executed following the EDF scheme. A slack time queue $\mathcal{STQ}$ is also maintained for each processor to keep track of the slack time(s) from (partially) canceled job(s) in it. Whenever the current job J_{ij} of task τ_i got chance to be executed, if it has been designated as the current floating redundant job of τ_i, its corresponding job in the other processor should be canceled immediately (because the floating redundant job does not need backup job) whose time budget should be inserted into the slack time queue $\mathcal{STQ}$ based on its deadline (line 28). Once the current floating redundant job J_{ij} is completed successfully, it is counted as an effective job and the next mandatory main/backup job after J_{ij} in the other processor should be designated as the new floating redundant job (lines 26-27). Otherwise in order to maintain the original (m,k)-constraint under fault tolerance all jobs following J_{ij} in the same window of k_i jobs should not be designated as floating redundant job and therefore should be executed in parallel with their corresponding jobs in the other processor (lines 12-18). For jobs more than k_i job patterns/positions after J_{ij}, since they are not within the range of the same window which J_{ij} belongs to, they will not be affected by the failure of J_{ij} at all and can be designated as the floating redundant job in turn again, similar to the case of the initial floating redundant job in the beginning (line 20).

Note that in the case when the current floating redundant job J_{ij} is found to have failed due to transient fault, since all mandatory jobs following J_{ij} in all windows containing J_{ij} cannot be designated as floating redundant job, totally m_i mandatory jobs after J_{ij} need to be executed concurrently with their corresponding jobs in the other processor. In this scenario in order to reduce the energy consumption further, the execution of the corresponding jobs in the other processor should be procrastinated as late as possible such that the overlapped executions of the jobs in the primary and the spare processors could be reduced (lines 16-17). Regarding that, the corresponding jobs in the other processor could be procrastinated by a time interval φ_i calculated based on the following theorem. For easy of presentation, we adopt the following notation, *i.e.*, $\lceil x \rceil^+$ to represent $(1+\lfloor x \rfloor)$ throughout this paper.

Theorem 1: Given task set $\mathcal{T}=\{\tau_1,\tau_2,...,\tau_N\}$ to be scheduled with Algorithm 1. Let all tasks be ordered by increasing value of D_i, all mandatory job deadlines can be guaranteed if any mandatory job J_{ij} of task τ_i is delayed by no more than

φ_i time units (called the *delay period* of task τ_i) if for any instant of time t:

$$\forall i \quad 1 \le i \le n \quad t \ge \varphi_i + \sum_{D_q \le t} \lceil \frac{m_q+1}{k_q} \lceil \frac{t-D_q}{T_q} \rceil^+ \rceil C_q \qquad (6)$$

and

$$\forall j < i \quad \varphi_j \le \varphi_i \qquad (7)$$

Proof: Use contradiction. Assuming at certain time point t', some mandatory job 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 mandatory jobs with arrival times or delayed starting times no earlier than t_0 and with deadlines less than or equal to t'. Since no job has arrival time or delayed starting time earlier than time 0, t_0 is well defined. We consider two cases:

- 1) At time t_0, there is no pending workload from mandatory jobs with delayed starting time and with deadlines less than or equal to t'. Then according to [15], the total mandatory work demand within the interval $[t_0, t']$ is bounded by $\sum_{D_q \le (t'-t_0)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_0-D_q}{T_q} \rceil^+ \rceil C_q$. Since some job missed the deadline at t', we have $\sum_{D_q \le (t'-t_0)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_0-D_q}{T_q} \rceil^+ \rceil C_q > (t'-t_0)$. On the other hand, considering the first busy interval, let $t = (t'-t_0)$, from Equation (6), we have $\sum_{D_q \le (t'-t_0)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_0-D_q}{T_q} \rceil^+ \rceil C_q \le (t'-t_0)$. Contradiction!
- 2) At time t_0, there is pending workload from mandatory jobs with delayed starting time and with deadlines less than or equal to t'. In this case the processor is idle at t_0^-. Let $t_1 (< t_0)$ be the latest time before t' when there are no pending mandatory jobs prior to t_1 with deadlines less than or equal to t'. The the mandatory work demand consumed in the interval $[t_0, t']$ is generated by the mandatory jobs arriving in the interval $[t_1, t']$. Obviously the mandatory work demand within the interval $[t_1, t']$ is bounded by $\sum_{D_q \le (t'-t_1)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_1-D_q}{T_q} \rceil^+ \rceil C_q$. Let k be the maximal index among the tasks with deadlines no larger than $(t'-t_1)$. Since there is deadline missing at t', we have

  $$\sum_{D_q \le (t'-t_1)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_1-D_q}{T_q} \rceil^+ \rceil C_q > (t'-t_0) \qquad (8)$$

  Note that the idle interval $[t_1, t_0]$ can only caused by the delay of certain task arriving at t_1, say τ_x, whose delay time is bounded by φ_x. Since τ_x also contribute to the work demand within $[t_1, t']$, from Equation (7), $\varphi_x \le \varphi_k$. So the idle interval length $(t_0 - t_1)$ is bounded by φ_k. Together with the result from Equation (8) we have $(t'-t_1) = (t_0-t_1) + (t'-t_0) \le \varphi_k + (t'-t_0) \le \varphi_k + \sum_{D_q \le (t'-t_1)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_1-D_q}{T_q} \rceil^+ \rceil C_q$. In Equation (6), letting $t = (t'-t_1)$, contradiction reached!

□

The rationale of Equation (6) is to find the maximal time φ_i before any absolute deadline of the mandatory jobs from τ_i that the work-demand of the mandatory jobs from τ_i and other mandatory job(s) with deadline(s) no later than it can be delayed such that no mandatory job deadline will be missed. Based on it, φ_i can be computed as

$$\varphi_i = \min\{d_i - (\sum_{D_q \le d_i} (\lceil \frac{m_q+1}{k_q} \lceil \frac{d_i-D_q}{T_q} \rceil^+ \rceil) C_q)\} \qquad (9)$$

for all $d_i \le L$, where L is the ending point of the first busy period when executing the mandatory jobs only and d_i is the absolute deadline of any mandatory job of task τ_i belonging to L. Note that, when calculating φ_i, if $\varphi_i < \varphi_j$ for any task τ_j with index less than τ_i, *i.e.*, $j < i$, the value of φ_j should be reset to be the same as φ_i due to condition (7) in Theorem 1.

Based on Algorithm 1, the estimation of the total energy consumption of task set $\mathcal{T}$ could be calculated as:

$$\begin{aligned} E &= \sum_i \frac{H(m_i+1)}{k_i} C_i + \sum_i \frac{H m_i}{k_i} \lambda(s_{max}) C_i \\ &+ 2P_{idle} H (1 - \frac{1}{2} \sum_i \frac{(m_i+1)C_i}{k_i P_i}) \end{aligned} \qquad (10)$$

where $\lambda(s_{max})$ is the average fault rate at the maximal processor speed s_{max}.

Note that the energy calculated above is indeed an upper bound of the energy consumption by Algorithm 1 because during execution, if some idle intervals are longer than t_{sd}, those idle intervals can be shutdown/wake-up dynamically to reduce actual energy consumption further (lines 32-37).

Moreover, during the runtime of Algorithm 1, at any time there are at most n mandatory main/backup jobs in its read queue. So the online complexity of Algorithm 1 is $O(n)$.

IV. Energy-constrained standby-sparing based on window transferring scheme

Although the floating redundant job scheme in Section III is quite helpful in estimating the required total energy consumption of the standby-sparing system and checking its feasibility under the given energy budget constraint, it needs to increase the m_i value of each task by 1 (to accommodate the extra mandatory job used as the floating redundant job), which might sometime affect the schedulability of the task set. This could be illustrated using the following example.

Consider another task set of two tasks, *i.e.*, $\tau_1 = (4,4,3,2,4)$, and $\tau_2 = (10,10,4,1,3)$, to be executed in a standby-sparing system with given energy budget constraint within its hyper period 240 to be 120 units. In order to apply algorithm 1, the task set needs to firstly increase the (m,k)-constraints of tasks τ_1 and τ_2 to be $(3,4)$ and $(2,3)$, respectively. However, it is easy to verify that the task set will not be schedulable under such new temporary (m,k)-constraints. On the other hand, to preserve the scheduability of the task set, if we apply the approach in [8] to execute the task set under the original (m,k)-constraints, although the task set is schedulable, the estimated total busy energy consumption will be 136 units, which has exceeded the given energy budget

constraint and therefore cannot ensure the feasibility of the task set.

However, if we follow a different way of scheduling the task set, it is still possible to ensure the feasibility of the task set. Before presenting our new approach, we need to define a variation of the E-pattern as followed. Based on it, the pattern π_{ij} for job J_{ij}, is defined as [23]:

$$\pi_{ij} = \begin{cases} \text{“1”} & \text{if } j = \lfloor \lceil \frac{(j-1+r_i)\times m_i}{k_i} \rceil \times \frac{k_i}{m_i} \rfloor + 1 \\ \text{“0”} & \text{otherwise} \end{cases} \quad j = 1,2,\cdots \tag{11}$$

Note that the above definition is actually a rotated version of the original E-pattern which can be regarded as rotating the E-pattern defined in Equation (4) to the right by r_i bits. For example, for a given (m,k)-constraint of $(3,6)$, its original E-patten is “101010”. If we rotate it to the right by $r_i = 1$ bit, the resulting patterns will be “010101” which are the same as defined according to Equation (11). For convenience, we call the pattern defined by (11) a rotation of the original E-pattern and represent it as E^{r_i}-pattern.

With the above definition, we have the following lemma.

Lemma 1: For any task τ_i with (m,k)-constraint of (m_i,k_i), let $y_i = m_i\frac{k_i+1}{m_i+1}$ if $\frac{k_i+1}{m_i+1}$ is an integer and $m_i < k_i$; or let $y_i = k_i - 1$ if $\frac{k_i-1}{2} \le m_i < k_i - 1$. Let $r_i = \lceil \frac{y_i - m_i}{m_i} \rceil$. Let the jobs of τ_i within each separate window of y_i jobs be partitioned with either E-pattern or E^{r_i}-pattern based on the new (m,k)-constraint of (m_i,y_i), its original (m,k)-constraint is satisfied.

Proof: According to Lemma 1, there are two possibilities for the value of y_i: if $\frac{k_i+1}{m_i+1}$ is an integer, then $y_i = m_i\frac{k_i+1}{m_i+1}$; or if $\frac{k_i-1}{2} \le m_i < k_i - 1$, $y_i = k_i - 1$. Under both possibilities y_i is an integer. When we inspect any two consecutive separate windows of y_i jobs in the resulting job patterns from Lemma 1, obviously there are two cases in general, (*i*) both windows are determined with same type of patterns; or (*ii*) they are determined with different type of patterns.

For case (*i*), without lose of generality, let's assume the case when the two consecutive windows of y_i jobs, namely window 1 and window 2, are partitioned with E-pattern and E^{r_i}-pattern, respectively, as shown in Figure 3. Then in Window 1, according to [24], the maximal number of consecutive “0”s is equal to r_i defined in Lemma 1, which happened at rightmost side of Window 1. Meanwhile, in window 2, since according to Definition (11) E^{r_i}-pattern is achieved by rotating E-pattern to the right by r_i bits, then in the leftmost side of Window 2 there are exactly r_i “0”s. Considering any sliding window of $(y_i + r_i)$ jobs starting from the current position of Window x (obviously in the beginning there are m_i “1”s in it), each time when we move window x to the right by one position, the number of “1”s in it will not change because the patterns for the leftmost $(y_i - r_i)$ jobs in Window 1 are the same as the rightmost $(y_i - r_i)$ jobs in Window 2, according to the definition of E^{r_i}-pattern. As such, until Window x reached the position of Window y, the number of “1”s in the sliding Window x is always m_i.

For case (*ii*), let's assume after Window 2, the next window, namely Window 3, has the same patterns as Window 2, *i.e.*,

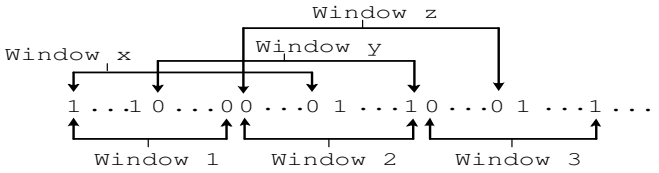


Fig. 3. The job patterns under consecutive windows.

E^{r_i}-pattern. Then obviously the patterns for the leftmost $(y_i - r_i)$ jobs in Window 3 are the same as the rightmost $(y_i - r_i)$ jobs in Window 1. So if we continue to move Window y to the right, until Window y reached the position of Window z, the number of “1”s in the sliding Window y will remain the same, *i.e.*, m_i. After that, if we continue move Window z to the right, obviously the number of “1”s in it will be no less than m_i, either. The case when both two consecutive windows are partitioned based on E-pattern is similar.

Based on the above statements, the resulting pattern from Lemma 1 can always satisfy the (m,k)-constraint of $(m_i,(y_i + r_i))$. Next we will show that $(y_i + r_i) = k_i$. We also check it under the two possibilities:

Possibility (*i*): $\frac{k_i+1}{m_i+1}$ is an integer. Since in this case $y_i = m_i\frac{k_i+1}{m_i+1}$, $\frac{y_i}{m_i} = \frac{k_i+1}{m_i+1}$ is an integer. So

$$\begin{aligned} y_i + r_i &= m_i\frac{k_i+1}{m_i+1} + \lceil \frac{y_i - m_i}{m_i} \rceil = m_i\frac{k_i+1}{m_i+1} + \lceil \frac{y_i}{m_i} \rceil - 1 \\ &= m_i\frac{k_i+1}{m_i+1} + \lceil \frac{k_i+1}{m_i+1} \rceil - 1 = m_i\frac{k_i+1}{m_i+1} + \frac{k_i+1}{m_i+1} - 1 \\ &= (m_i+1)\frac{k_i+1}{m_i+1} - 1 = k_i \end{aligned} \tag{12}$$

Possibility (*ii*): $\frac{k_i-1}{2} \le m_i < k_i - 1$. Since in this case $y_i = k_i - 1$,

$$\frac{k_i-1}{2} \le m_i < k_i - 1 \Leftrightarrow 1 < \frac{k_i-1}{m_i} \le 2 \Leftrightarrow 1 < \frac{y_i}{m_i} \le 2 \tag{13}$$

As such, in this case

$$r_i = \lceil \frac{y_i - m_i}{m_i} \rceil = \lceil \frac{y_i}{m_i} \rceil - 1 = 2 - 1 = 1 \tag{14}$$

So,

$$y_i + r_i = (k_i - 1) + 1 = k_i \tag{15}$$

From the above, for both possibilities, $(y_i + r_i) = k_i$. □

To help understand Lemma 1, let us consider a task τ_i with (m,k)-constraint of (3,7). According to Lemma 1, $y_i = 6$ and $r_i = 1$. Then based on Equation (11), E^1 =“010101”. From Lemma 1, one possible pattern for task τ_i is “101010010101010101$\cdots$”. It is easy to verify that it can satisfy the original (m,k)-constraint of (3,7).

Note that Lemma 1 effectively sets up a straightforward way of converting a window-constraint of m_i/y_i (within each separate window of y_i jobs) to the original (m,k)-constraint of (m_i,k_i). It is similar to, but tighter than, the result in [14] which can convert a window constraint of $m_i/\frac{(m_i+k_i)}{2}$ to the original (m,k)-constraint of (m_i,k_i). For example, for the above task τ_i with (m,k)-constraint of (3,7), in order to satisfy its original

(m,k)-constraint, based on Lemma 1 it only needs to satisfy the window-constraint of 3/6 in each separate window of 6 jobs whereas according to the approach in [14], it needs to satisfy the window-constraint of (3,5) in each separate window of 5 jobs. Obviously the former one is easier to be schedulable than the latter one. In the following, we will formulate this result into a lemma.

Lemma 2: For any task τ_i, if both the window constraints of $m_i/(m_i\frac{k_i+1}{m_i+1})$ and $m_i/\frac{(m_i+k_i)}{2}$ can be used to define τ_i's job patterns successfully under E-pattern, the job patterns determined based on the former one has better schedulability than the job patterns determined based on the latter one.

Proof: Since m_i value is the same, to prove $m_i/m_i\frac{k_i+1}{m_i+1}$ has better schedulability than $m_i/\frac{(m_i+k_i)}{2}$, we only need to prove:

$$\begin{aligned} m_i\frac{k_i+1}{m_i+1} \geq \frac{(m_i+k_i)}{2} \quad &\Leftrightarrow \quad 2(k_i+1)m_i \geq (m_i+k_i)(m_i+1) \\ &\Leftrightarrow \quad 2k_im_i+2m_i \geq m_i^2+m_i+m_ik_i+k_i \\ &\Leftrightarrow \quad (1-m_i)(m_i-k_i) \geq 0 \qquad (16) \end{aligned}$$

Which is true because $(m_i \geq 1)$ and $(k_i \geq m_i)$. □

Based Lemma 1, our new approach of scheduling the task set with the given energy budget constraint can be described as followed: for each task τ_i, if $\frac{k_i+1}{m_i+1}$ is an integer and $m_i < k_i$ or $\frac{k_i-1}{2} \leq m_i < k_i-1$, we let y_i and r_i be determined according to Lemma 1. Then base on it we can determine the mandatory main jobs of task τ_i in one processor with E-pattern and their backup jobs in the other processor with E^{r_i}-pattern, both based on the window constraint of m_i/y_i first. Since $r_i \geq 1$ if $m_i < k_i$ or $\frac{k_i-1}{2} \leq m_i < k_i-1$, in any separate window of y_i jobs, each mandatory main job and its backup job in the other processor are not in the same time frame. In other words, they are totally shifted way. As such, if any mandatory main job is completed successfully, its backup job can be canceled entirely. Even if the mandatory main job were found to have failed upon completion, its backup job can still be executed timely. In the worst case, if all mandatory main jobs in a separate window of y_i jobs have failed, their backup jobs in the other processor will all need to be executed. In this scenario the resulting job pattern will be equivalent to case (i) in the proof of Lemma 1. Then according to Lemma 1, its original (m,k)-constraint will be satisfied.

Particularly, for tasks τ_1 and τ_2 in the above example task set, their corresponding window constraints will be $2/3$ and $1/2$, respectively. Then based on them the mandatory main jobs of tasks τ_1 and τ_2 are determined under E-pattern and they can be scheduled in different processors, as shown in Figure 4. Meanwhile, the backup jobs for τ_1 and τ_2 will be determined based on E^{r_i}-pattern and can be reserved in different processors as well. As such, since each mandatory main job and its backup job are totally shifted away, once a mandatory main job (for example, J_{11}) is completed successfully, its backup job (*i.e.*, $J_{12}^{'}$) in the other processor could be canceled entirely. If any mandatory main job of task τ_i had failed, its corresponding backup job in the other processor could still be invoked and executed timely (for example, if the main job J_{17} in the primary processor had failed, its backup job $J_{18}^{'}$ could still be executed timely in the spare processor, as shown in Figure 4(b)). In this way, even in the worst case that all mandatory main jobs in one window had failed, as stated above, its original (m,k)-constraint can still be ensured. Following the same rationale, if we assume the probability of transient fault to be 10^{-5}, then the expected energy consumption of all backup jobs within one window of y_i jobs for all tasks will be $(3\times2+4\times1)\times10^{-5}=0.0001$. Therefore the estimated total busy energy consumption subject to faults before the hyper period 240 will be 90+0.0001= 90.0001 units, which is much less than the given energy budget constraint. If we assume the idle power $P_{idle}=0.05$, the total energy consumption should be calculated based on Equation (17), which will be 105.6001 units, still below the given energy budget constraint and therefore feasible.

From the above example we can see that there is great potential for meeting the given energy budget constraint by determining the mandatory main jobs and their backup jobs based on E-pattern and E^{r_i}-pattern, respectively (which can satisfy the original (m,k)-constraint according to Lemma 1). Based on the above principles, our standby-sparing scheduling scheme based on window transferring is presented in Algorithm 2.

As shown in Algorithm 2, in the beginning, for each task $\tau_i \in \mathcal{T}$, if $\frac{k_i+1}{m_i+1}$ is an integer and $m_i < k_i$ or $\frac{k_i-1}{2} \leq m_i < k_i-1$ (how to handle the case when these conditions are not met will be discussed in next section), we firstly determine the values of y_i and r_i according to Lemma 1 and re-partition task τ_i and its backup task $\tau_i^{'}$ with E-pattern and E^{r_i}-pattern (both based on its new temporary QoS constraint of m_i/y_i), respectively. Note that task τ_i or its backup task $\tau_i^{'}$ can be executed in either the primary processor or the spare processor, without affecting their schedulablility. As such, for any mandatory main job J_{ij}, its backup job (denoted as $\tilde{J}_{ij}$) will be the job $J_{i(j+r_i)}^{'}$ of its backup task $\tau_i^{'}$ (line 1). Similar to Algorithm 1, during runtime, in both the primary and the spare processors, a mandatory job ready queue (MQ) and a slack time queue $\mathcal{STQ}$ are maintained. Upon arrival, a job of task τ_i is inserted into the MQ only when its job pattern is "1". All jobs in MQ will be executed according to the EDF scheme. When the current job J_{ij} of task τ_i got chance to be executed, if J_{ij} is a mandatory main job, it should be executed as soon as possible and the slack time in the $\mathcal{STQ}$, if available, should be reclaimed to facilitate its early completion (line 9); otherwise it should be executed as late as possible (line 11).

Note that, when the current mandatory main job J_{ij} is completed successfully, whether its backup job in the other processor should be canceled or not needs to be handled carefully. Specifically, if job J_{ij} is within the same time frame of the backup job of some other job, its backup job cannot be canceled. For example, in Figure 4, assuming J_{17} in the primary processor had failed, then its backup job $J_{18}^{'}$ in the spare processor needed to be executed. Meanwhile, in the primary processor, the mandatory main job J_{18} was executed

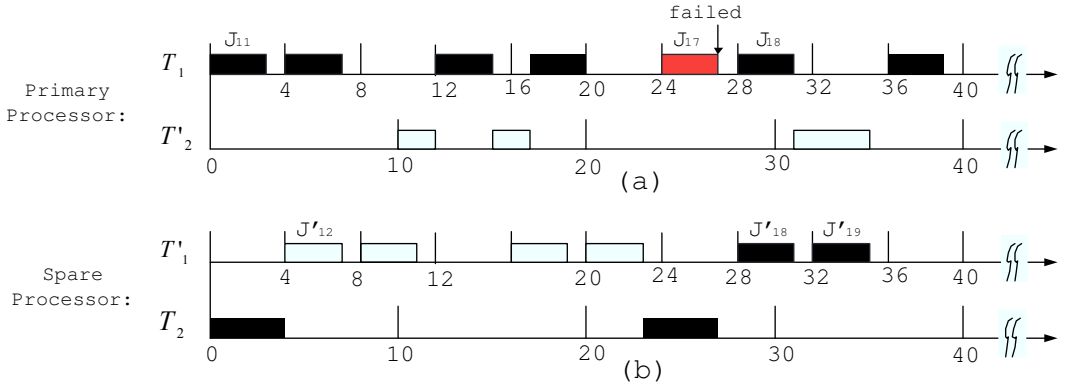


Fig. 4. The schedule for the mandatory main/bakcup jobs based on window transferring scheme (a) in the primary processor; (b) in the spare processor.

Algorithm 2 The algorithm based on window transferring

1: **Preparations:** For each task $\tau_i \in \mathcal{T}$, if $\frac{k_i+1}{m_i+1}$ is an integer and $m_i < k_i$ or $\frac{k_i-1}{2} \leq m_i < k_i - 1$, determine y_i and r_i according to Lemma 1. Re-partition τ_i and its backup task $\tau_i^{'}$ with the new temporary QoS constraint of m_i/y_i based on E-pattern and E^{r_i}-pattern, respectively. For any mandatory main job J_{ij}, mark job $J_{i(j+r_i)}^{'}$ in the other processor as its backup job (denoted as $\tilde{J}_{ij}$);
2:
3: **For either the primary processor or the spare processor:**
4:
5: **Upon the execution of a mandatory job J_{ij} at time t_{cur}:**
6: Execute J_{ij} following the EDF scheme;
7: **if** any slack time $STQ_i(t)$ with earlier deadline than J_{ij} is available **then**
8: **if** J_{ij} is a mandatory main job **then**
9: Reclaim the slack time to execute J_{ij} as soon as possible;
10: **else**
11: Use the slack time to procrastinate J_{ij} as late as possible;
12: **end if**
13: **end if**
14:
15: **Upon the completion of a job J_{ij} at current time t_{cur}:**
16: **if** the execution of job J_{ij} is successful **then**
17: Let $J_{ij}^{'}$ be the job in the other processor within the same time frame as J_{ij};
18: **if** $J_{ij}^{'}$ is not a mandatory main/backup job **then**
19: Cancel J_{ij}'s backup job $\tilde{J}_{ij}$, *i.e.*, $J_{i(j+r_i)}^{'}$ in the other processor entirely and add its time budget to the slack queue $\mathcal{STQ}$;
20: **else**
21: Cancel the remaining part of $J_{ij}^{'}$ and add its residue time budget to the slack queue $\mathcal{STQ}$;
22: **end if**
23: Repeat lines 32-37 in Algorithm 1
24: **end if**

in the same time frame as $J_{18}^{'}$. Suppose J_{18} was completed successfully. In this case, if we had canceled its backup job $J_{19}^{'}$ in the spare processor, then in the time interval [24,36] there would be only one valid job because J_{18} and $J_{18}^{'}$ were in the same time frame and would effectively generate only one valid job. Consequently the window constraint of $2/3$ will be violated in this interval and the original (m,k)-constraint of $(2,4)$ will be violated in the time interval of [20,36].

As mentioned, the main reason for the above problem is that, in Algorithm 2, due to the pattern rotation, all mandatory main jobs and their backup jobs are shifted away into different time frames. As a result it is possible that within the current time frame the execution of the current mandatory main job could be overlapped with that of the backup job of some other mandatory job (for example, J_{18} and $J_{18}^{'}$ in Figure 4). When that happened, they effectively contributed only one valid job to the window they belong to. As such, if the backup job of the current mandatory main job is canceled, the number of valid jobs in the same window will decrease by 1 which could cause the QoS constraint in it to be violated and subsequently cause the original (m,k)-constraint to be violated as well. Therefore, in this case, even if the current mandatory main job is completed successfully, its backup job should not be canceled, as implied in line 18 of Algorithm 2.

Similar to the upper bound of the energy consumption calculated in Section III, based on Algorithm 2, if for each and every task $\tau_i \in \mathcal{T}$, $\frac{k_i+1}{m_i+1}$ is an integer and $m_i < k_i$ or $\frac{k_i-1}{2} \leq m_i < k_i - 1$, an upper bound of the total energy consumption of task set $\mathcal{T}$ could be calculated as:

$$E = \sum_i \frac{Hm_i}{y_i} C_i + \sum_i \frac{Hm_i}{y_i} C_i \lambda(s_{max}) + 2P_{idle} H (1 - \frac{1}{2} \sum_i \frac{m_i C_i}{y_i P_i}) \tag{17}$$

where y_i is determined according to Lemma 1.

Note that the worst case in Algorithm 2 happens when at certain point, all mandatory main jobs in one separate window have failed consecutively and all their backup jobs in the other processor need to be executed, which will be equivalent to one of the scenarios in Lemma 1. Then according to Lemma 1, its original (m,k)-constraint can be ensured.

Similar to Algorithm 1, the online complexity of Algorithm 2 is also $O(n)$. Moreover, we have the following theorem.

Theorem 2: Given task set $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_N\}$ to be scheduled with Algorithm 2 in a standby-sparing system with

total energy budget of $\bar{E}_c$ within its hyper period, the system is feasible if: (*i*) for each and every task $\tau_i \in \mathcal{T}$, $\frac{k_i+1}{m_i+1}$ is an integer and $m_i < k_i$ or $\frac{k_i-1}{2} \leq m_i < k_i - 1$; (*ii*) $\mathcal{T}$ is schedulable with the (m,k)-constraint of each task τ_i in it replaced by (m_i, y_i), where y_i is determined according to Lemma 1; and (*iii*) the energy consumption E calculated based on Equation (17) does not exceed $\bar{E}_c$.

Proof: If for any task $\tau_i \in \mathcal{T}$ $\frac{k_i+1}{m_i+1}$ is an integer and $m_i < k_i$, then $\frac{k_i+1}{m_i+1} \geq 2$. So $r_i = \lceil \frac{y_i}{m_i} \rceil - 1 = \frac{k_i+1}{m_i+1} - 1$ is also an integer and $r_i \geq 1$. If $\frac{k_i-1}{2} \leq m_i < k_i - 1$, from Lemma 1 $r_i = 1$. Thus in either case Algorithm 2 can be applied. The main issue is to ensure the original (m,k)-constraint. The worst case in Algorithm 2 happens when at certain point, all mandatory main jobs in one separate window have failed consecutively, then all their backup jobs in the other processor need to be executed, which will be equivalent to one of the scenarios in Lemma 1. Then according to Lemma 1, its original (m,k)-constraint can be assured. □

V. Integrated approach based on hybrid schemes

Although the above window transferring scheme in Algorithm 2 could be more efficient than the floating redundant job scheme in Section IV in meeting the given energy budget constraint, the main issue for it is that, for tasks which do not satisfy the conditions in line 1 of Algorithm 2, they will not be able to be transferred in this way. On the other hand, the floating redundant job scheme in Section IV also has the issue that it might affect the schedulability of the task set because it needs to have one more mandatory job reserved for each task. Regarding that, in order to still meet the energy budget constraint while respecting the schedulability of the task set, the best way is to partition the original task set $\mathcal{T}$ into three parts and schedule them with the schemes in Section IV, Section III, and the regular job procrastination scheme similar to lines 13-18 in algorithm 1, respectively, in an integrated approach. Correspondingly, Problem 1 could be reformulated as follows:

Problem 2: Given system $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_N\}$, partition the original task set $\mathcal{T}$ into three subsets, *i.e.*, $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ to be scheduled with the window transferring scheme in Algorithm 2, floating redundant job scheme in Algorithm 1, and the regular job procrastination scheme, respectively such that the estimated total energy consumption does not exceed the given energy budget constraint $\bar{E}_c$ while satisfying the (m,k)-constraints for all tasks under the fault tolerant requirement.

In order to solve Problem 2, in this paper we proposed a heuristics based on "branch-and-bound", which is presented in Algorithm 3.

From Algorithm 3, our approach determines task by task if each task $\tau_i \in \mathcal{T}$ should be scheduled with the window transferring scheme in Section IV, the floating redundant job scheme in Section III, or the regular job procrastination scheme. When Algorithm 3 is finished, it is possible to reach certain hybrid configuration in which the tasks in subsets $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$ are partitioned based on the QoS constraint of $m_i/(m_i\frac{k_i+1}{m_i+1})$ or $m_i/(k_i-1)$, (m_i+1,k_i), and (m_i,k_i) to be scheduled with the window transferring scheme in Algorithm 2, floating redundant job scheme in Algorithm 1, and the job procrastination scheme following lines 13-18 in Algorithm 1, respectively. And the resulting configuration should be the one with the minimum estimated total energy consumption E_{total} computed in line 28. Once the final E_{total} is calculated, we will compare it with the given energy constraint E_c. If $E_{total} \leq E_c$, the task set is guaranteed to be feasible. Otherwise the feasibility of the task set cannot be guaranteed.

Note that after the original task set $\mathcal{T}$ was divided into three subsets $\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, the calculation of the delay period of φ_i for each task τ_i under the hybrid configuration should be updated

Algorithm 3 Task set partitioning using Branch-and-Bound.

1: **Input:** task set $\mathcal{T}$ with original (m,k)-constraints;
2: **Output:** task set $\tilde{\mathcal{T}} = \mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z}$, where $\mathcal{X}$, $\mathcal{Y}$ and $\mathcal{Z}$ are the subsets to be scheduled with the schemes in Section IV, Section III, and the regular job procrastination scheme, respectively;
3: $\mathcal{X} = \emptyset$;
4: $\mathcal{Y} = \emptyset$;
5: $\mathcal{Z} =$ all tasks in $\mathcal{T}$;
6: Sort the tasks in $\mathcal{Z}$ according to non-increasing order of $\frac{m_i C_i}{k_i P_i}, i = 1,..,n$;
7: $\tilde{\mathcal{T}} = \mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z}$;
8: $E_{total} = E_{bound} = 2(\sum_i \{C_i \frac{Hm_i}{k_i}\} + P_{idle} H(1 - \sum_i \frac{m_i C_i}{k_i P_i}))$;
9: //The estimated total energy consumption using standby-sparing for all mandatory main/bakcup jobs based on the original (m,k)-constraints without energy management;
10: SS-Partition ($\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, $\tilde{\mathcal{T}}$, E_{bound});
11: output ($\tilde{\mathcal{T}}$);
12:
13: **FUNCTION SS-Partition**($\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, $\tilde{\mathcal{T}}$, E_{bound})
14: **for** each task $\tau_i \in \tilde{\mathcal{T}}$ **do**
15: **if** $\frac{k_i+1}{m_i+1}$ is an integer and $m_i < k_i$ or $\frac{k_i-1}{2} \leq m_i < k_i - 1$ **then**
16: Determine y_i according to Lemma 1;
17: Set τ_i's new temporary QoS constraint to be m_i/y_i;
18: $\mathcal{X} = \mathcal{X} \cup \{\tau_i\}$;
19: **else**
20: Set τ_i's new temporary QoS constraint to be (m_i+1, k_i);
21: $\mathcal{Y} = \mathcal{Y} \cup \{\tau_i\}$;
22: **end if**
23: Remove τ_i from $\mathcal{Z}$;
24: **if** $\mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z}$ is schedulable **then**
25: Compute the energy consumption E_X for all mandatory jobs in $\mathcal{X}$ based on Equation (17);
26: Compute the energy consumption E_Y for all mandatory jobs in $\mathcal{Y}$ based on Equation (10);
27: Compute the energy consumption E_Z for all mandatory jobs in $\mathcal{Z}$ based on Equation (5);
28: $E_{total} = E_X + E_Y + E_Z$;
29: **if** $E_{total} < E_{bound}$ **then**
30: $E_{bound} = E_{total}$;
31: $\tilde{\mathcal{T}} = \mathcal{X} \cup \mathcal{Y} \cup \mathcal{Z}$;
32: **end if**
33: SS-Partition ($\mathcal{X}$, $\mathcal{Y}$, $\mathcal{Z}$, $\tilde{\mathcal{T}}$, E_{bound});
34: **else**
35: Restore τ_i's QoS constraint to its original (m_i,k_i)-constraint and put it back to $\mathcal{Z}$;
36: **end if**
37: **end for**

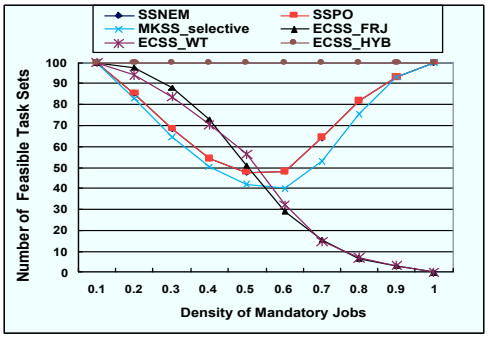
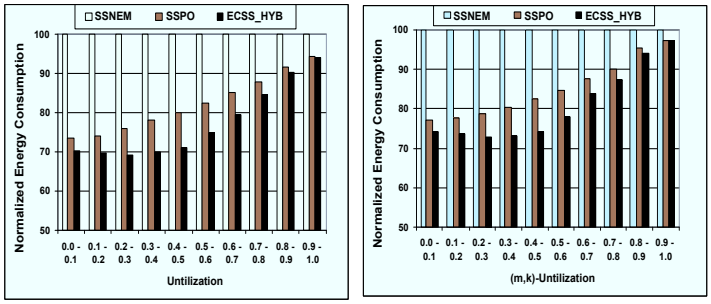


Fig. 5. Feasibility comparison of the different approaches.



(a) (b)

Fig. 6. Actual energy subject to (a) No faults; (b) System faults.

as followed.

$$\varphi_i = \min\{d_i \quad - \sum_{D_x \le d_i}^{\tau_x \in \mathcal{X}} (\lceil \frac{m_x}{y_x} \lceil \frac{d_i - D_x}{T_x} \rceil^+ \rceil) C_x \\ - \sum_{D_y \le d_i}^{\tau_y \in \mathcal{Y}} (\lceil \frac{(m_y+1)}{k_y} \lceil \frac{d_i - D_y}{T_y} \rceil^+ \rceil) C_y \\ - \sum_{D_z \le d_i}^{\tau_z \in \mathcal{Z}} (\lceil \frac{m_z}{k_z} \lceil \frac{d_i - D_z}{T_z} \rceil^+ \rceil) C_z\} \qquad (18)$$

for all $d_i \le L$, where d_i is the absolute deadline of a mandatory job of task τ_i and L is the ending point of the first busy period when executing the mandatory jobs, and $\forall j < i \ \ \varphi_j \le \varphi_i$.

VI. Evaluation

In this section, we evaluate the performance of our approach by comparing with the existing approaches in literature. Specifically, the performance of six different approaches were studied:

- *SSNEM* The task sets were partitioned with E-pattern, and the mandatory jobs in the primary and the spare processors were executed concurrently without delay.
- *SSPO* The task sets were partitioned with E-pattern to satisfy the given (m,k)-constraints. Then the mandatory jobs were scheduled with the preference oriented scheme in [8] but without applying DVS.
- $MKSS_{Selective}$ The task sets were scheduled with the approach from [20] but based on EDF scheme. The task set were partitioned with deeply-red pattern first to satisfy the given (m,k)-constraints. Then the selective approach in [20] was applied.
- $ECSS_{FRJ}$ This is our approach purely based on the floating redundant job scheme proposed in Section III.
- $ECSS_{WT}$ This is our approach purely based on the window transferring scheme proposed in Section IV.
- $ECSS_{HYB}$ This is our hybrid approach proposed in Section V.

The periodic task sets in our experiments consists of five to ten tasks with the periods randomly chosen in the range of $[5, 50]ms$. The m_i and k_i for the (m,k)-constraint were also randomly generated such that k_i was uniformly distributed between 2 to 10, and $1 \le m_i \le k_i$. The worst case execution time (WCET) of a task was also uniformly distributed. We assume the processor idle power $P_{idle} = 0.05$ and minimal shut-down interval $t_{sd} = 2ms$.

Firstly, we inspected the feasibility of the different approaches under different density of mandatory jobs. The density of mandatory jobs, defined as $\frac{1}{N}\sum_i \frac{m_i}{k_i}$, was divided into intervals of length 0.1 each of which contained at least 5000 task sets generated. Based on it we checked the feasibility of the task sets when scheduled by the different approaches. We assumed the maximal energy budget constraint is randomly picked from [1.5X, 2.5X], where X is the energy consumption for executing the mandatory jobs of all tasks under their original (m,k)-constraints within the hyper period in one processor without energy management. The numbers of feasible task sets were normalized to that by $ECSS_{HYB}$. The results are shown in Figure 5. From Figure 5, it is not hard to see that, in all cases, $ECSS_{HYB}$ always has the best feasibility. Moreover, for different density of mandatory jobs, the other approaches presented different performance on feasibilities. As can be seen, when the density of mandatory jobs is very small, *i.e.*, close to 0.1, the total number of task sets feasible by the other approaches were all very close to that by $ECSS_{HYB}$. However, with the increase of density of the mandatory jobs, the feasibility of the different approaches became much different. For $ECSS_{FRJ}$ and $ECSS_{WT}$, their feasibilities were always decreasing because $ECSS_{FRJ}$ needed to increase the value of m_i by 1 while $ECSS_{WT}$ needed to reduce the window size from k_i to $y_i = m_i \frac{k_i+1}{m_i+1}$, both can affect the schedulability of the task sets. On the other hand, the feasibilities of *SSNEM*, *SSPO*, and $MKSS_{Selective}$ decreased fast first but then became close to $ECSS_{HYB}$ again when the density of mandatory jobs were relatively high, for example, larger than 0.8. This is because, when the density of mandatory jobs is high, the hybrid approach in $ECSS_{HYB}$ might not be able to partition plenty of tasks to be scheduled under Algorithm 1 or Algoirthm 2 due to schedulability constraint. Instead in this case most tasks can only be scheduled under the regular job procrastination scheme whose estimated total energy consumption is the same as *SSPO*. However, as shown in Figure 5, when the density of mandatory jobs is moderate, for example, between 0.3 and 0.7, the feasibility of $ECSS_{HYB}$ is much better than the other approaches, with maximal improvement of nearly 60%, mainly due to its capability of combining the advantages of the different schemes under the hybrid configuration. On the other hand, the feasibility of *SSNEM* and *SSPO* overlapped with each other completely because both of them are based on E-pattern. So their schedulabilities were the same (their estimated total energy consumptions were also equal to each other,

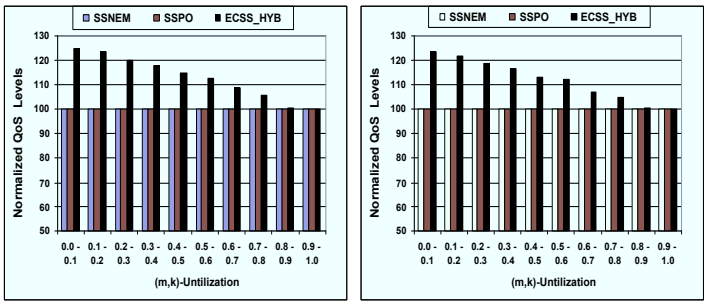


Fig. 7. The QoS subject to (a) No faults; (b) System faults.

as discussed earlier). It is also noted that the feasibility of $MKSS_{Selective}$ is lower than that by *SSNEM* and *SSPO* mainly because it is based on deeply-red pattern whose schedulability is not as good as E-pattern [22].

Next, we inspected the actual energy consumption of the different approaches. With system feasibility in mind, this time we mainly compared our proposed approach with the most typical one in the previous approaches, *i.e.*, *SSPO* which is the previous approach with the best feasibility. Moreover, since according to the above results, the feasibilities of $ECSS_{FRJ}$ and $ECSS_{WT}$ are much worse than the other approaches when the density of the mandatory jobs were relatively high, we did not include them in this part of test, either. Also considering the impact of workloads on energy performance, we checked the actual energy consumption of the different approaches based on the system utilization, *i.e.*, $\sum_i \frac{m_i C_i}{k_i P_i}$ which was divided into intervals of length 0.1 and each interval contains at least 20 task sets feasible or at least 1000 task sets generated. We conducted two sets of tests.

In the first set, we checked the energy performance when no fault occurred during the hyper period. The results were normalized to that by *SSNEM* and shown in Figure 6(a).

From Figure 6(a), it is easy to see that even when all approaches were feasible, both the approaches with energy management, *i.e.*, $ECSS_{HYB}$ and *SSPO* still consumed much less actual energy than the approach without energy management, *i.e.*, *SSNEM*. Moreover, the actual energy consumption of $ECSS_{HYB}$ is much lower than *SSPO* in most intervals. For example, when the system workload is moderate, the actual energy consumed by $ECSS_{HYB}$ can be around 22% less than that by *SSPO*. The main reason is that, under this scenario, by adopting the hybrid approach in Section V, $ECSS_{HYB}$ can help minimize the overlapped execution between the mandatory jobs and their backup jobs in two processors more efficiently. Moreover, for those tasks that cannot be applied with the window transferring scheme or the floating redundant job scheme, letting them be applied with the job procrastination scheme with delay intervals calculated in Equation (18) also helped save energy consumption effectively.

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 [2] by assuming Poisson distribution with an average fault rate of 10^{-5}. The result is shown in Figure 6(b).

As could be seen, under this scenario, the actual energy consumption by our new approach, *i.e.*, $ECSS_{HYB}$ is still much less than the previous approaches. The actual energy reduction by $ECSS_{HYB}$ over *SSPO* can be up to 20%. This is also because of the capability of $ECSS_{HYB}$ in scheduling the tasks with the hybrid configuration as mentioned above. Additionally, when fault(s) occurred, procrastinating the backup jobs within the same window of the failed job using the delay intervals calculated in Equation (18) also contributed to part of the energy savings due to its capability of shifting the executions of the mandatory main job(s) and their backup job(s) when necessary.

Finally, with the QoS in mind, we also inspected the QoS levels that the different approaches could provide when all approaches were feasible. The QoS level was defined as the ratio of the number of effective jobs over the total number of jobs within the hyperperiod. We also conducted two sets of tests.

In the first set, we checked the QoS when no fault occurred within the hyperperiod. The results were normalized to that by *SSNEM* and shown in Figure 7(a). From Figure 7(a), we can see that our newly proposed approach, *i.e.*, $ECSS_{HYB}$ could provide much better QoS levels than the previous approaches. Compared with *SSNEM* and *SSPO*, the maximal QoS improvement could be around 25%. This is because, different from *SSNEM* and *SSPO* which could only provide a minimum set of jobs that "just" satisfied the (m,k)-constraints, $ECSS_{HYB}$, by adopting hybrid configurations, could have extra number of jobs scheduled under the floating redundant job scheme or the window transferring scheme. Therefore it could generally accommodate more valid jobs in its schedule, generating better QoS levels.

In the second set, we assumed the system could be subject to permanent and/or transient faults with same fault rate as in the previous group of test. The result is shown in Figure 7(b).

From Figure 7(b), the QoS improvement subject to faults by our newly proposed approach, *i.e.*, $ECSS_{HYB}$ over the previous approaches is quite similar to that when no fault ever occurred, for the same reasons as stated above.

VII. Conclusion

QoS, fault tolerance, and energy budget constraint are among the primary concerns in the design of real-time embedded systems. In this paper, we firstly presented two novel scheduling algorithms which can ensure feasibility for the standby-sparing systems under tighter energy budget constraint than the traditional ones: one adopting floating redundant job scheme and one adopting window transferring scheme. Then based on the aforementioned constraints a hybrid approach was proposed to achieve better performance. Through extensive evaluations, our results demonstrated that the proposed techniques significantly outperformed the existing state of the art approaches in terms of feasibility, energy saving, and QoS performance for weakly hard real-time systems while ensuring fault tolerance under given energy budget constraint.

ACKNOWLEDGE*

This work is partly supported by the U.S. NSF under grants HRD 1800403, CNS/SaTC 2039583, and HRD 1828811, and by the DoD Center of Excellence in AI and Machine Learning (CoE-AIML) at Howard University under Contract Number W911NF-20-2-0277 with the U.S. Army Research Laboratory.

REFERENCES

[1] A. Taherin, M. Salehi, and A. Ejlali, "Reliability-aware energy management in mixed-criticality systems," *IEEE Transactions on Sustainable Computing*, vol. 3, no. 3, pp. 195–208, July 2018.

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

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

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] S. Safari, S. Hessabi, and G. Ershadi, "Less-mics: A low energy standby-sparing scheme for mixed-criticality systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pp. 1–1, 2020.

[11] M. Ansari, A. Yeganeh-Khaksar, S. Safari, and A. Ejlali, "Peak-power-aware energy management for periodic real-time applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 779–788, 2020.

[12] L. Niu and G. Quan, "Peripheral-conscious energy-efficient scheduling for weakly hard realctime systems," *International Journal of Embedded Systems*, vol. 7, no. 1, pp. 11–25, 2015.

[13] 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.

[14] 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.

[15] 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.

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

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

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

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

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

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

[22] 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.

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

[24] L. Niu, "Energy efficient scheduling for real-time systems with qos guarantee," *Journal of Real-Time Systems*, vol. 47, no. 2, pp. 75–108, 2011.