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

LINWEI NIU, Howard University, USA
DANDA B. RAWAT, Howard University, USA
JONATHAN MUSSELWHITE, Howard University, USA
ZONGHUA GU, Umeå University, Sweden
QINGXU DENG, Northeastern University, China

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 both periodic and aperiodic tasks in a weakly hard real-time environment. The standby-sparing systems adopt a primary processor and a spare processor to provide fault tolerance for both permanent and transient faults. For such kind of systems, we firstly propose several novel standby-sparing schemes for the periodic tasks which can ensure the system feasibility under tighter energy budget constraint than the traditional ones. Then based on them integrated approachs for both periodic and aperiodic tasks are proposed to minimize the aperiodic response time whilst achieving better energy and QoS performance under the given energy budget constraint. 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 the given energy budget constraint.

CCS Concepts: • **Computer systems organization → Real-time systems**; **Embedded systems**; *reliability*; *Redundancy*.

Additional Key Words and Phrases: energy constraint, standby-sparing, quality of service, fault tolerance, real-time scheduling

ACM Reference Format:
Linwei Niu, Danda B. Rawat, Jonathan Musselwhite, Zonghua Gu, and Qingxu Deng. 20XX. Energy-Constrained Scheduling for Weakly Hard Real-Time Systems Using Standby-Sparing. *ACM Trans. Des. Autom. Electron. Syst.* , , Article 111 (20XX), 35 pages. https://doi.org/XXXXXXX.XXXXXXX

1 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 [34] or other portable embedded

Authors' addresses: Linwei Niu, linwei.niu@howard.edu, Howard University, the Department of Electrical Engineering and Computer Science, Washington, DC, USA, 20059; Danda B. Rawat, Howard University, the Department of Electrical Engineering and Computer Science, Washington, DC, USA, 20059, Danda.Rawat@howard.edu; Jonathan Musselwhite, Howard University, 2400 Sixth Street NW, Washington, DC, USA, 20059, jonathan.musselwhite@bison.howard.edu; Zonghua Gu, Umeå University, the Department of Applied Physics and Electronics, Umeå, , Sweden, 90187, zonghua.gu@umu.se; Qingxu Deng, Northeastern University, the School of Computer Science and Engineering,, Shenyang, Liaoning, China, 110819, dengqx@mail.neu.edu.cn.

1084-4309/20XX/-ART111 $15.00
https://doi.org/XXXXXXX.XXXXXXX

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 [44]. Generally, computing system faults can be classified into permanent faults and transient faults [16]. 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. [37, 43]) 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 based on time redundancy, *i.e.*, to reserve recovery jobs whenever possible, to tolerate transient faults through re-execution of the faulty jobs. 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 [9, 11, 14, 36]. 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 [11]. 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. [9, 11, 14, 36]) have been reported to reduce energy by letting the executions of the main jobs and their corresponding 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 [32]. When considering the chip thermal effect, peak-power-aware standby-sparing techniques utilizing energy management schemes were presented in [3].

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 this end, 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) [24]. Two well known weakly hard QoS models are the (m, k)-model [13] and the *window-constrained* model [38]. 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.* [28] 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 [40], 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 without recovery. 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 energy consumption of the main/backup jobs often needs to be estimated for the worst case because their actual energy consumption will usually remain unknown offline and cannot be accurately predictable, which could make the estimation of the total energy consumption go beyond the given energy budget constraint unnecessarily.

In many real-time applications such as multimedia and telecommunication systems, both periodic tasks and aperiodic tasks are required in which periodic tasks are time driven with (m, k)-deadlines while aperiodic tasks are event driven with soft deadlines [8]. For such kind of mixed task systems, two design objectives need to be achieved: *(i)* the (m, k)-constraints of the periodic tasks must be ensured at any time; *(ii)* the response time of the aperiodic tasks should be minimized. In this paper, we will add a third objective to it, *i.e.*, the given energy budget constraint in the mission cycle should never be exceeded. Based on them, we study the problem of energy constrained standby-sparing for both periodic and aperiodic tasks in a weakly hard real-time environment 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 2 presents the preliminaries. Section 3 presents our approaches for purely periodic tasks. Section 4 presents our approaches for mixed systems containing both periodic and aperiodic tasks. In Section 5, we present our evaluation results. In Section 6, we discuss the related work. In Section 7, we offer our conclusions.

2 PRELIMINARIES

2.1 System model

The real-time system T considered in this paper contains a number of periodic tasks, *i.e.*, $\{\tau_1, \tau_2, \cdots, \tau_N\}$, scheduled according to the earliest deadline first (EDF) scheduling scheme. Each periodic 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, all in milliseconds. 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.

The system T can also contain a number of aperiodic tasks, *i.e.*, $\{\tau_{N+1}, \tau_{N+2}, \cdots, \tau_{N+M}\}$. Each aperiodic task is characterized using two parameters, *i.e.*, (C_i, D_i), which represent the worst case execution time and the soft deadline for it.

We assume the task set is to be executed in a standby-sparing system with a limited energy budget/supply of $\bar{E}_B$ 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 periodic tasks, *i.e.*, $H = LCM(k_i P_i)$ 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.

2.2 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 [22] 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 (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), *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*.

2.3 Fault Model

Similar to the standby-sparing systems in [11, 14, 30], 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 [26]. Assume that the energy and the timing overheads of sanity (or consistency) checks are E_{sc} and t_{sc}, respectively. Moreover, following the fault model in [44], we assume that the transient faults will present Poisson distribution [39] and the average transient fault rate for systems running at the maximal speed s_{max} (and the corresponding supply voltage) is $\sigma(s_{max})$. Based on it, the average job fault rate for task τ_i at the maximal processor speed s_{max}, represented as $\lambda_i(s_{max})$ can be calculated as:

$$\lambda_i(s_{max}) = (1 - e^{-\sigma(s_{max})C_i}) \tag{4}$$

Also for permanent faults, we follow the model adopted in [30] that if a permanent fault occurs on any of the cores, the other core can still execute one copy of each task's instances. However, when a permanent fault occurs, the system loses its capability of tolerating any additional (transient or permanent) faults until the faulty core is repaired or replaced [30].

3 ENERGY-CONSTRAINED STANDBY-SPARING FOR PURELY PERIODIC TASKS

3.1 Approach based on floating redundant job scheme

For the scheduling of periodic tasks in a weakly hard real-time system, one essential part is to determine the *mandatory* jobs in them to be scheduled under standby-sparing. Two well-known partitioning strategies are the *evenly distributed pattern* (or E-pattern) [28] 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):

$$\pi_{ij} = \begin{cases} \text{“1”} & \text{if } j = \lfloor \lceil \frac{(j-1)\times m_i}{k_i} \rceil \times \frac{k_i}{m_i} \rfloor \\ \text{“0”} & \text{otherwise} \end{cases} \qquad j = 1, 2, 3, \cdots \tag{5}$$

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

$$\pi_{ij} = \begin{cases} \text{“1”} & \text{if } 1 \leq j \bmod k_i \leq m_i \\ \text{“0”} & \text{otherwise} \end{cases} \qquad j = 1, 2, 3, \cdots \tag{6}$$

The mandatory/optional job partitioning according to equation (5) has the property that it spreads out the mandatory jobs *evenly* in each task along the time. Moreover, it is shown in [23] 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.

Note that the job patterns defined with either E-pattern or R-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

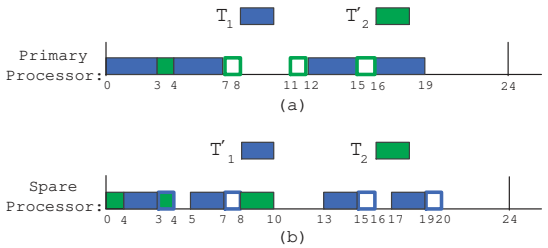


Fig. 1. The schedule for the mandatory main/bakcup jobs under the preference oriented scheme in [11]: (a) in the primary processor; (b) in the spare processor.

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. Obviously, the energy consumption in such kind of standby-sparing systems could be quite considerable. In order to reduce energy consumption, in [14], 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 [6] such that, once the main tasks/jobs are completed successfully, their corresponding backup tasks/jobs could be (partially) canceled. In [11], a more advanced technique named preference oriented scheme was adopted which, in both the primary and the backup processors, lets some tasks be scheduled under EDS scheme while the other tasks be scheduled under EDL scheme. In [25], an energy-aware approach based on the execution of optional jobs was proposed for task sets partitioned under deeply-red pattern [17] which is weaker than E-pattern in ensuring the schedulability of the task sets [23]. Although the approaches in [11, 14, 25] are able to reduce the actual energy consumption of the standby-sparing system to some extent, since none of them could predict the quantifiable amount of energy that can be saved in advance, the total energy budget still has to be estimated using the summation of the worst case energy consumption in both processors, *i.e.*

$$\sum_i \frac{Hm_i}{k_i P_i}(C_i + E_{sc}) + P_{idle}H(1 - \sum_i \frac{m_i(C_i + t_{sc})}{k_i P_i}) + \sum_i \frac{Hm_i}{k_i P_i}C_i + P_{idle}H(1 - \sum_i \frac{m_i C_i}{k_i P_i}) \quad (7)$$

where H is the hyper period.

Otherwise if the given energy budget constraint $\bar{E}_B$ during the hyper period is less than the energy consumption estimated with Equation (7), 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 energy budget constraint $\bar{E}_B$. This could be illustrated using the following examples.

Consider a task set consisting of two tasks, *i.e.*, $\tau_1 = (2.9, 4, 4, 6)$, and $\tau_2 = (1.9, 8, 8, 2, 3)$, to be executed in a standby-sparing system with given energy budget constraint $\bar{E}_B = 28$ units within its hyper period 24 millisecond, assuming $P_{idle} = 0.05$, $E_{sc} = 0.2$, and $t_{sc} = 0.1$ millisecond.

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 [11] (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 energy consumption to be what is calculated using Equation (7). Based on it, the estimated worst case energy consumption is 32.95

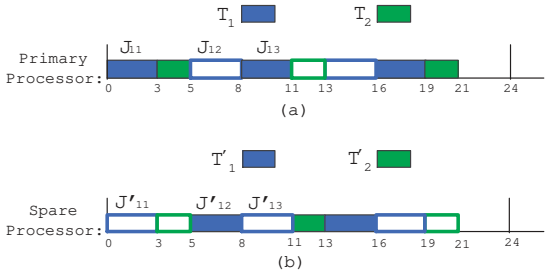


Fig. 2. The schedule for the mandatory main/bakcup jobs under the floating redundant job scheme: (a) in primary processor; (b) in spare processor.

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 21.8 units. Even when we have the energy consumption during the idle period included, the estimated total energy consumption of the system within the hyper period will be (21.8 + $P_{idle} \times 27$=) 23.15 units, which is less than $\bar{E}_B$ 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} per millisecond, then the expected energy consumption of all backup jobs within one window of k_i jobs for task τ_1 and task τ_2 will be 0.001682 and 0.0002166 units, respectively. After adding it to the above result, the total estimated energy consumption of the system subject to fault(s) will be 23.1518986 units, which is still less than the given energy budget constraint and therefore feasible.

Algorithm 1 The algorithm based on floating redundant job

1: **Preparations:** For each task $\tau_i \in T$, 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 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 mandatory 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 S;
29: **else**
30: Cancel J_{ij}'s corresponding job in the other processor and add its residue time budget to the slack queue 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**=0

Note that in the above approach the mandatory main/backup 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 $\tau_i \in T$, 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 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 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 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 3.1. *Given periodic tasks $T = \{\tau_1, \tau_2, ..., \tau_N\}$ 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:*

$$\forall i \;\; 1 \le i \le N \quad t \ge \varphi_i + \sum_{D_q \le t} \lceil \frac{m_q + 1}{k_q} \lceil \frac{t - D_q}{T_q} \rceil^+ \rceil (C_q + t_{sc}) \tag{8}$$

and

$$\forall j < i \;\; \varphi_j \le \varphi_i \tag{9}$$

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 [28], the total mandatory work demand within the interval $[t_0, t']$ is bounded by $\sum_{D_q \le (t'-t_0)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_0-D_q}{T_q} \rceil^+ \rceil (C_q + t_{sc})$. Since some job missed the deadline at t', we have $\sum_{D_q \le (t'-t_0)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_0-D_q}{T_q} \rceil^+ \rceil (C_q + t_{sc}) > (t' - t_0)$. On the other hand, considering the first busy interval, let $t = (t' - t_0)$, from Equation (8), we have $\sum_{D_q \le (t'-t_0)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_0-D_q}{T_q} \rceil^+ \rceil (C_q + t_{sc}) \le (t' - t_0)$. 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 $\sum_{D_q \le (t'-t_1)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_1-D_q}{T_q} \rceil^+ \rceil (C_q + t_{sc})$. 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

$$\sum_{D_q \le (t'-t_1)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_1-D_q}{T_q} \rceil^+ \rceil (C_q + t_{sc}) > (t' - t_0) \tag{10}$$

  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 (9), $\varphi_x \le \varphi_k$. So the idle interval length $(t_0 - t_1)$ is bounded by φ_k. Together with the result from Equation (10) we have $(t' - t_1) = (t_0 - t_1) + (t' - t_0) \le \varphi_k + (t' - t_0) \le \varphi_k + \sum_{D_q \le (t'-t_1)} \lceil \frac{m_q+1}{k_q} \lceil \frac{t'-t_1-D_q}{T_q} \rceil^+ \rceil (C_q + t_{sc})$. In Equation (8), letting $t = (t' - t_1)$, contradiction reached!

□

The rationale of Equation (8) is to find the maximal time φ_i before any absolute deadline of the mandatory jobs from τ_i to which 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

$$\varphi_i = \min\{d_i - (\sum_{D_q \le d_i} (\lceil \frac{m_q+1}{k_q} \lceil \frac{d_i - D_q}{T_q} \rceil^+ \rceil)(C_q + t_{sc}))\} \tag{11}$$

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 (9) in Theorem 3.1.

Based on Algorithm 1, the estimation of the total energy consumption of a system consisting of purely periodic tasks could be calculated as:

$$\begin{aligned} E &= \sum_i \frac{H(m_i+1)}{k_i P_i}(C_i + E_{sc}) + \sum_i \frac{H(m_i+1)}{k_i P_i} \lambda_i(s_{max}) m_i C_i \\ &+ 2P_{idle} H(1 - \frac{1}{2} \sum_i \frac{(m_i+1)(C_i + t_{sc})}{k_i P_i}) \end{aligned} \tag{12}$$

where $\lambda_i(s_{max})$ is the average job fault rate for task τ_i at the maximal processor speed s_{max}.

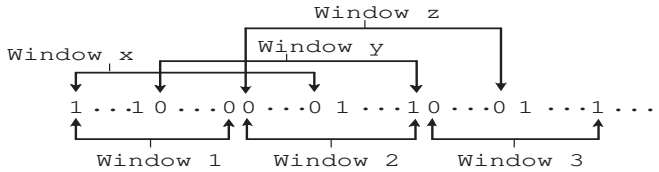


Fig. 3. The job patterns under consecutive windows.

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 ready queue. So the online complexity of Algorithm 1 is $O(N)$.

3.2 Approach based on window transferring scheme

Although the floating redundant job scheme in Section 3.1 is quite helpful in estimating the required total energy consumption of the system and checking its feasibility under the given energy budget constraint, it needs to increase the m_i value of each periodic task by 1 (to accommodate the extra mandatory job used as the floating redundant job), which might affect the schedulability of the task set. This could be illustrated using the following example.

Consider another task set consisting of two periodic tasks, *i.e.*, $\tau_1 = (2.9, 4, 4, 2, 4)$, and $\tau_2 = (3.9, 10, 10, 1, 3)$, to be executed in a standby-sparing system with given energy budget constraint within its hyper period 240 to be 200 units, assuming same value of P_{idle}, E_{sc}, and t_{sc} as in the previous example. 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 [11] to execute the task set under the original (m, k)-constraints, although the task set is schedulable, the estimated total energy consumption based on Equation (7) will be 255.99 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 that, 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 [27]:

$$\pi_{ij} = \begin{cases} \text{“1”} & \text{if } j = \lfloor \lceil \frac{(j-1+r_i)\times m_i}{k_i} \rceil \times \frac{k_i}{m_i} \rfloor + 1 \\ \text{“0”} & \text{otherwise} \end{cases} \qquad j = 1, 2, \cdots \tag{13}$$

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 (5) 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 (13). For convenience, we call the pattern defined by (13) 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 3.2. *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 $\frac{k_i-1}{2} \leq m_i < k_i - 1$. 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 3.2, 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 $\frac{k_i-1}{2} \leq m_i < k_i - 1$, $y_i = k_i - 1$. 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 3.2, obviously there are two cases in general, (*i*) the two windows are determined with different type of patterns; or (*ii*) they are determined with same 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 [21], the maximal number of consecutive "0"s is equal to r_i defined in Lemma 3.2, which happened at the rightmost side of Window 1. Meanwhile, in window 2, since according to Definition (13) 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.*, 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 3.2 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

$$\begin{aligned} y_i + r_i &= m_i\frac{k_i+1}{m_i+1} + \lceil\frac{y_i - m_i}{m_i}\rceil = m_i\frac{k_i+1}{m_i+1} + \lceil\frac{y_i}{m_i}\rceil - 1 \\ &= m_i\frac{k_i+1}{m_i+1} + \lceil\frac{k_i+1}{m_i+1}\rceil - 1 = m_i\frac{k_i+1}{m_i+1} + \frac{k_i+1}{m_i+1} - 1 \\ &= (m_i+1)\frac{k_i+1}{m_i+1} - 1 = k_i \end{aligned} \tag{14}$$

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

$$\frac{k_i-1}{2} \le m_i < k_i - 1 \Leftrightarrow 1 < \frac{k_i-1}{m_i} \le 2 \Leftrightarrow 1 < \frac{y_i}{m_i} \le 2 \tag{15}$$

As such, in this case

$$r_i = \lceil\frac{y_i - m_i}{m_i}\rceil = \lceil\frac{y_i}{m_i}\rceil - 1 = 2 - 1 = 1 \tag{16}$$

So,

$$y_i + r_i = (k_i - 1) + 1 = k_i \tag{17}$$

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

To help understand Lemma 3.2, consider a task τ_i with (m, k)-constraint of (3,7). According to Lemma 3.2, $y_i = 6$ and $r_i = 1$. Then based on Equation (13), E^1 ="010101". From Lemma 3.2, 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 3.2 effectively sets up a straightforward way of converting a window-constraint of m_i/y_i [1] (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 [38] 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 3.2 it only needs to satisfy the window-constraint of 3/6 in each separate window of 6 jobs whereas according to the approach in [38], 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 as well.

Algorithm 2 The algorithm based on window transferring

1: **Preparations:** For each task $\tau_i \in T$, 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 3.2. 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 mandatory 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 the backup job of a failed mandatory main 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 STQ;
20: **else**
21: Cancel the remaining part of $J^{'}_{ij}$ and add its residue time budget to the slack queue STQ;
22: **end if**
23: Repeat lines 32-37 in Algorithm 1
24: **end if**=0

LEMMA 3.3. *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.*

[1] Here we follow the notation used in [38] which used the notation "x/y" to indicate the window-constraint requiring within each separate window of y jobs at least x jobs out of them must meet their deadlines.

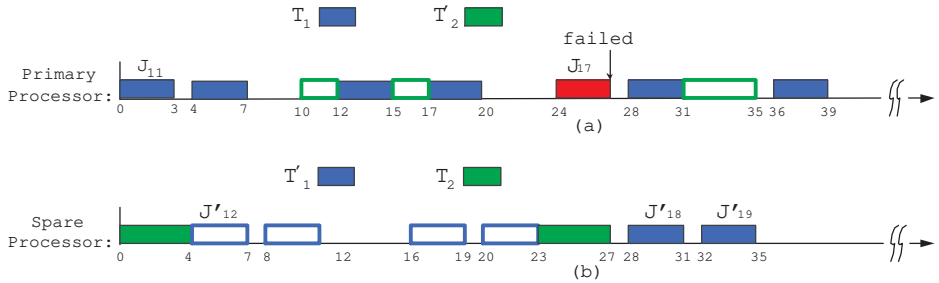


Fig. 4. The schedule for the mandatory main/bakcup jobs based on window transferring scheme in (a) primary processor; (b) spare processor.

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:

$$
\begin{aligned}
m_i\frac{k_i+1}{m_i+1} \geq \frac{(m_i+k_i)}{2} \quad &\Leftrightarrow \quad 2(k_i+1)m_i \geq (m_i+k_i)(m_i+1) \\
&\Leftrightarrow \quad 2k_im_i+2m_i \geq m_i^2+m_i+m_ik_i+k_i \\
&\Leftrightarrow \quad (1-m_i)(m_i-k_i) \geq 0 \qquad (18)
\end{aligned}
$$

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

Based Lemma 3.2, 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 3.2. 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 3.2. Then according to Lemma 3.2, 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 transient fault rate to be 10^{-5} per millisecond, then the expected energy consumption of all backup jobs within one window of y_i jobs for task τ_1 and task τ_2 will be 0.006728 and 0.0018252, respectively. With energy during the processor idle time under $P_{idle} = 0.05$ included, the total energy consumption within

the hyper period will be 189.06855 units, which is 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 3.2). 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 $\tau_i \in T$, 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 3.2 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 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 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 failed 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 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 the time interval of [24,36] 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 the backup job of some other failed mandatory main 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 can not be canceled, as implied in line 18 of Algorithm 2.

Similar to the upper bound of the energy calculated in Section 3.1, based on Algorithm 2, if for each and every periodic task τ_i in the system, $\frac{k_i+1}{m_i+1}$ 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 a system consisting of purely periodic tasks could

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

Input: task set T consisting of purely periodic tasks with original (m, k)-constraints;
Output: task set $\tilde{T} = X \cup Y \cup Z$, where X, Y and Z are the subsets to be scheduled with the schemes in Section 3.2, Section 3.1, and the regular job procrastination scheme, respectively;
$X = \emptyset$;
$Y = \emptyset$;
Z = all tasks in T;
Sort the tasks in Z according to non-increasing order of $\frac{m_i C_i}{k_i P_i}$, $i = 1, .., n$;
$\tilde{T} = X \cup Y \cup Z$;
$E_{total} = E_{bound} = \sum_i \frac{Hm_i}{k_i P_i}(C_i + E_{sc}) + P_{idle} H(1 - \sum_i \frac{m_i(C_i + t_{sc})}{k_i P_i}) + \sum_i \frac{Hm_i}{k_i P_i} C_i + P_{idle} H(1 - \sum_i \frac{m_i C_i}{k_i P_i})$;
//The estimated total energy consumption using standby-sparing for all mandatory main/bakcup jobs based on the original (m, k)-constraints without energy management;
SS-Partition $(X, Y, Z, \tilde{T}, E_{bound})$;
output $(\tilde{T})$;

FUNCTION SS-Partition$(X, Y, Z, \tilde{T}, E_{bound})$
for each task $\tau_i \in \tilde{T}$ **do**
  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**
    Determine y_i according to Lemma 3.2;
    Set τ_i's new temporary QoS constraint to be m_i/y_i;
    $X = X \cup \{\tau_i\}$;
  else
    Set τ_i's new temporary QoS constraint to be $(m_i + 1, k_i)$;
    $Y = Y \cup \{\tau_i\}$;
  end if
  Remove τ_i from Z;
  if $X \cup Y \cup Z$ is schedulable **then**
    Compute the energy consumption E_X for all mandatory jobs in X based on Equation (19);
    Compute the energy consumption E_Y for all mandatory jobs in Y based on Equation (12);
    Compute the energy consumption E_Z for all mandatory jobs in Z based on Equation (7);
    $E_{total} = E_X + E_Y + E_Z$;
    if $E_{total} < E_{bound}$ **then**
      $E_{bound} = E_{total}$;
      $\tilde{T} = X \cup Y \cup Z$;
    end if
    SS-Partition $(X, Y, Z, \tilde{T}, E_{bound})$;
  else
    Restore τ_i's QoS constraint to its original (m_i, k_i)-constraint and put it back to Z;
  end if
end for=0

be calculated as:

$$\sum_i \frac{Hm_i}{y_i P_i}(C_i + E_{sc}) + \sum_i \frac{Hm_i}{y_i p_i} \lambda_i(s_{max}) m_i C_i + 2P_{idle} H(1 - \frac{1}{2} \sum_i \frac{m_i(C_i + t_{sc})}{y_i P_i}) \quad (19)$$

where y_i is determined according to Lemma 3.2.

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 3.2. Then according to Lemma 3.2, 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 3.4. *Given a system consisting of purely periodic tasks $\{\tau_1, \tau_2, \cdots, \tau_N\}$ to be scheduled with Algorithm 2 in a standby-sparing system with total energy budget of $\bar{E}_B$ within its hyper period, the system is feasible if: (*i*) for each and every task τ_i in the system, $\frac{k_i+1}{m_i+1}$ is an integer and $m_i < k_i$ or $\frac{k_i-1}{2} \leq m_i < k_i - 1$; (*ii*) the system 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 3.2; and (*iii*) the energy consumption E calculated based on Equation (19) does not exceed $\bar{E}_B$.*

PROOF. If for any task τ_i in the system $\frac{k_i+1}{m_i+1}$ 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 3.2 $r_i = 1$. 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 case (i) in the proof of Lemma 3.2. Then according to Lemma 3.2, its original (m,k)-constraint can be assured. □

3.3 Integrated approach based on combined schemes

Although the above window transferring scheme in Algorithm 2 could be more efficient than the floating redundant job scheme in Section 3.2 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 3.2 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 into three parts and schedule them with the schemes in Section 3.2, Section 3.1, and the regular job procrastination scheme similar to lines 13-18 in algorithm 1, respectively, in an integrated approach. Correspondingly, the problem to be solved could be formulated as follows:

PROBLEM 1. *Given task set consisting of purely periodic tasks $\{\tau_1, \tau_2, \cdots, \tau_N\}$, partition the original task set into three subsets,* i.e.*, X, Y, and 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 $\bar{E}_B$ while satisfying the (m,k)-constraints for all tasks under the fault tolerant requirement.*

To solve Problem 1, 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 $\tau_i \in T$ should be scheduled with the window transferring scheme in Section 3.2, the floating redundant job scheme in Section 3.1, or the regular job procrastination scheme. When Algorithm 3 is finished, it is possible to reach certain combined configuration in which the tasks in subsets X, Y, 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 $\bar{E}_B$. If $E_{total} \leq \bar{E}_B$, 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 T was divided into three subsets X, Y, Z, the calculation of the delay period of φ_i in Equation (11) for each task τ_i under the combined configuration should be updated as followed.

$$\begin{aligned}\varphi_i = \min\{d_i \quad &- \sum_{D_x \le d_i}^{\tau_x \in X} (\lceil \frac{m_x}{y_x} \lceil \frac{d_i - D_x}{T_x} \rceil^+ \rceil)(C_x + t_{sc}) \\ &- \sum_{D_y \le d_i}^{\tau_y \in Y} (\lceil \frac{(m_y + 1)}{k_y} \lceil \frac{d_i - D_y}{T_y} \rceil^+ \rceil)(C_y + t_{sc}) \\ &- \sum_{D_z \le d_i}^{\tau_z \in Z} (\lceil \frac{m_z}{k_z} \lceil \frac{d_i - D_z}{T_z} \rceil^+ \rceil)(C_z + t_{sc})\}\end{aligned} \tag{20}$$

3.4 Improving the QoS by executing optional jobs

It is not hard to see that for the tasks in either subset X or subset Y, there are redundant jobs in them. So their quality of service (QoS) will be better than the ones based on their corresponding original (m, k)-patterns. For the tasks in subset Z which are partitioned based on the original (m, k)-patterns, in order to improve their QoS, we can choose to execute some optional jobs when no mandatory job is pending for execution. However, since the execution of the optional jobs could potentially cause the overall energy consumption of the system to exceed the energy consumption estimated using Equation (7), we cannot execute the optional jobs arbitrarily. Instead, when we are about to execute (some of) the optional jobs, we should adopt the following criteria in choosing the eligible optional jobs for execution.

It is not hard to see that, for any particular task τ_i in Z, in order to guarantee its overall energy consumption within the hyper period will not exceed what is calculated using Equation (7), we must ensure that, within each sliding window of k_i jobs from task τ_i, there are no more than $2m_i$ jobs executed in it in total. Then the problem is how to guarantee that? Obviously, it is not practical to check all window of k_i jobs in task τ_i during execution. In order to solve the problem, here we will propose a more efficient approach which only needs to check a limited number of windows at runtime. Specifically, at runtime, we only need to guarantee that, for all sliding windows of k_i jobs containing τ_i's current job J_{ij}, there are no more than $2m_i$ jobs (to be) executed in each of them. In this way, the total energy consumption can always be bounded by the energy consumption calculated using Equation (7) (in later part of this section we will formulate that into a theorem, *i.e.*, Theorem 3.5 and provide the formal proof for it as well). With that in mind, we can revise the algorithm for scheduling the tasks in Z correspondingly. The details of it are presented in Algorithm 4.

As shown in Algorithm 4, whenever no mandatory job is pending for execution and some optional job J_{ij} of task $\tau_i \in Z$ becomes available, we will check all sliding windows containing it to see if there are already $2m_i$ jobs executed in any of them. It is easy to see that, among such kind of windows, the oldest one should be the window starting with job $J_{i(j-(k_i-1))}$ and ending with job J_{ij} while the latest one should be the window starting with job J_{ij} and ending with job $J_{i(j+(k_i-1))}$. The optional job J_{ij} is eligible only when each and any of the sliding windows between the above oldest and latest windows contains less than $2m_i$ jobs in it (Lines 17-20). Moreover, since the optional jobs always have lower priorities than the mandatory ones and could be preempted by any of them, we also need to check if the optional jobs could be completed by its deadline and the earliest arrival time of the upcoming mandatory jobs, whichever smaller. Only the optional jobs that can satisfy the above requirements should be chosen as eligible ones. If there are multiple optional jobs of the tasks in Z becoming available simultaneously, the ties could be broken based on first-come-first-serve or by

Algorithm 4 The enhanced scheduling algorithm for the tasks in Z with optional job execution

For either the primary processor or the spare processor:

Upon the arrival of job J_{ij} of task $\tau_i \in Z$ at current time t_{cur}:
if J_{ij} is an optional job **then**
  Executable = **true**;
  for $x = (j - (k_i - 1))$ to j **do**
    NJ = 0; // NJ is the total number of jobs executed or to be executed in the current sliding window
    for $y = 0$ to $(k_i - 1)$ **do**
      if $J_{i(x+y)}$ is an optional job **then**
        if $J_{i(x+y)}$ has been executed **then**
          NJ = NJ +1;
        end if
      else
        NJ = NJ +2;
      end if
    end for
    if $(NJ \times (E(C_i) + E_{sc}) + E_{cw}(k_i)) \geq (2 \times m_i \times (E(C_i) + E_{sc}))$ **then**
      // $E(C_i)$ is the energy consumption of executing a job of task τ_i and $E_{cw}(k_i)$ is the energy overhead of checking windows for the current optional job J_{ij}. The value of $E_{cw}(k_i)$ can be achieved through a stored look-up table based on the value of k_i.
      Executable = **false**;
      Break;
    end if
  end for
  Let NTA be the earliest arrival time of the next upcoming mandatory job in the same processor;
  if (Executable == **true**) **then**
    if $(\min(NTA, d_{ij}) - t_{cur}) > (C_i + t_{sc})$ **then**
      Execute job $J_{i(x+y)}$ non-preemptively;
    end if
  else if $(NTA - t_{cur}) > t_{sd}$ **then**
    Shut down the processor and set up the wake-up timer to be $(NTA - t_{cur})$;
  end if
else
  // J_{ij} is a mandatory job
  if J_{ij} is a mandatory main job **then**
    Execute J_{ij} following the EDF scheme;
    if any slack time $STQ_i(t)$ with higher priority than J_{ij} is available **then**
      Reclaim the slack time to execute J_{ij} as soon as possible;
    end if
  else
    Revise r_{ij} to $\max\{(r_{ij} + \varphi_i), (t_{cur} + STQ_i(t_{cur}))\}$;
    Execute J_{ij} following the EDF scheme;
  end if
end if

Upon the completion of job J_{ij} of task $\tau_i \in Z$:
if the execution of job J_{ij} is successful **then**
  if J_{ij} is an optional job **then**
    Shift the future job patterns of τ_i correspondingly;
  else
    Cancel J_{ij}'s corresponding job in the other processor and add its residue time budget to the slack queue S;
  end if
end if=0

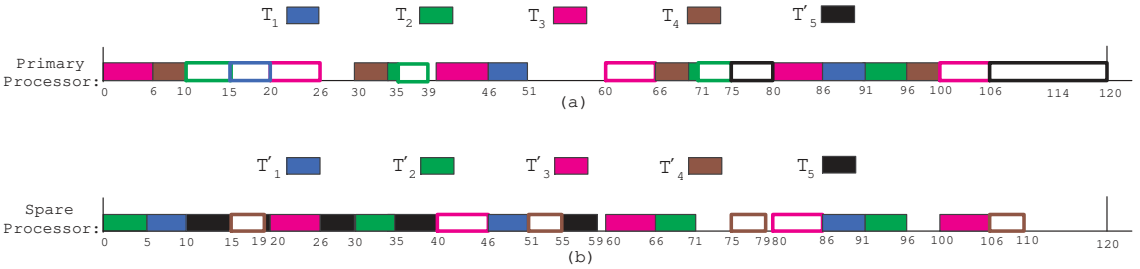


Fig. 5. (a) The schedule for the periodic main tasks $\tau_1 = (5, 20, 20, 3, 6)$, $\tau_2 = (5, 15, 15, 2, 4)$, $\tau_3 = (6, 10, 10, 2, 6)$, $\tau_4 = (4, 15, 15, 1, 3)$ based on E-pattern only for the tasks in Z and the aperiodic backup task $\tau_5^{'}$ in the primary processor; (b) The schedule for the periodic backup tasks $\tau_1^{'}$, $\tau_2^{'}$, $\tau_3^{'}$, and $\tau_4^{'}$ based on E-pattern only for the tasks in Z and the aperiodic main task $\tau_5 = (19, 120)$ in the spare processor.

randomly choosing one of them. Once chosen, the optional job should be executed non-preemptively to guarantee that it could be completed timely (Line 25). If the execution of the optional job J_{ij} is successful, the future job patterns for its owner task τ_i should be adjusted by shifting its future job patterns to the right correspondingly.

The online complexity of Algorithm 4 mainly comes from checking all the sliding windows containing the current optional job J_{ij}, which is at most $O(k_i^2)$. Since k_i is usually a small integer, the complexity of Algorithm 4 is suitable for online use in general.

THEOREM 3.5. *With Algorithm 4, the total energy consumption for the tasks in Z will not exceed the energy calculated with Equation (7).*

PROOF. Given any task τ_i in Z, for any arbitrary sliding window of k_i jobs (represented as $\bar{W}_i$) in it, we consider two cases: (i) There is no optional jobs executed in $\bar{W}_i$. Then based on the property of the (m, k)-pattern, there are exactly m_i mandatory jobs executed in $\bar{W}_i$ in one processor, thus totally $2m_i$ mandatory jobs (including both the mandatory main jobs and their backup ones) in the two processors altogether; (ii) There are some optional jobs executed in $\bar{W}_i$. In this case, from lines 4-26 of Algorithm 4, whenever any of the optional jobs executed in $\bar{W}_i$ became current, $\bar{W}_i$ would have been checked dynamically to ensure that there would be at most $2m_i$ jobs (to be) executed in it. Thus in both cases, there could be no more than $2m_i$ jobs executed in $\bar{W}_i$. Based on the arbitrarity that $\bar{W}_i$ was chosen, the conclusion of Theorem 3.5 follows. □

4 ENERGY-CONSTRAINED STANDBY-SPARING FOR BOTH PERIODIC AND APERIODIC TASKS

The scheduling for mixed task systems containing both periodic and aperiodic tasks is more complex as it must be able to ensure the (m, k)-deadlines for the periodic tasks while minimizing the average response times for the aperiodic tasks with soft deadlines. The scheduling algorithm for a mixed task set must also accomplish these goals without compromising the feasibility of the whole system under the given energy budget constraint. In order to do so, we can adopt a hierarchical priority assignment strategy, *i.e.*, letting the mandatory jobs and optional jobs for the periodic tasks be always executed in the upper priority band and lower priority band, respectively, while letting the aperiodic tasks be executed in the middle priority band. In this way, the execution of the aperiodic tasks will never interfere with that of the mandatory jobs of the periodic tasks. Therefore the (m, k)-deadlines for the periodic tasks can be assured. Meanwhile, the execution of the optional jobs will not impact the response time of the aperiodic tasks, either. Based on it, the scheduling schemes proposed

in Section 3 can still be applied to schedule the mandatory jobs and optional job of the periodic tasks in the upper priority band and lower priority band, respectively. The only difference is, when estimating the total energy consumption of the whole system, the energy calculation should include the energy consumption of the aperiodic tasks as well. In particular, when partitioning the task set using branch-and-bound, line 28 in Algorithm 3 should be updated as follows:

$$E_{total} = E_X + E_Y + E_Z + \sum_{q=N+1}^{N+M} \{C_q(1 + \lambda_q(s_{max})) + E_{sc}\} \tag{21}$$

where C_q represents the execution time of the aperiodic task τ_q.

Note that in the above equation, when calculating the energy consumption of the aperiodic tasks, we only need to include one fault free copy of each aperiodic task τ_q because since the aperiodic tasks do not require hard deadlines, we can always wait until the completion of its main job to determine whether it is necessary or not to invoke its backup job in the other processor.

Then the problem is **how to minimize the average response time of the aperiodic tasks**? Note that a general strategy to achieve this goal is to rearrange the periodic tasks in X to let all mandatory main jobs in it be executed in the primary processor while all backup jobs of them be executed in the spare processor. Meanwhile, let the main job(s) of the aperiodic task(s) be executed in the spare processor while their backup jobs be executed in the primary processor. In this way, since in the spare processor the backup jobs of X are most likely to be canceled, it will provide more chance for the main job(s) of the aperiodic task(s) to be completed earlier there. Moreover, by considering different types of pattern assignment in determining the mandatory jobs of the periodic tasks in the subset Z, additional reduction on the aperiodic response time could be achieved, which could be illustrated using the following examples.

As known, for the previous combined scheme in Section 3.3 we adopt E-patterns in determining the mandatory jobs of the periodic tasks in the subset Z because E-patterns tend to provide better schedulability than R-patterns in general [23]. However, when the aperiodic tasks are incorporated, this approach might not always be able to achieve the minimal aperiodic response time.

Consider a mixed task set consisting of four periodic tasks, *i.e.*, $\tau_1 = (5, 20, 20, 3, 6)$, $\tau_2 = (5, 15, 15, 2, 4)$, $\tau_3 = (6, 10, 10, 2, 6)$, $\tau_4 = (4, 15, 15, 1, 3)$, and an aperiodic task, *i.e.*, $\tau_5 = (19, 120)$, to be executed in a standby-sparing system with given energy budget constraint $\bar{E}_B = 160$ units within its hyper period 120. Then based on the window transferring strategy in Section 3.2 the mandatory main jobs of task τ_4 can be determined based on the window-constraint of $1/2$ first (based on it the original (m, k)-constraint of τ_4 can be satisfied according to Lemma 3.2). So $X = \{\tau_4\}$. Note that here τ_3 cannot adopt the same strategy because the value of $m_i \frac{k_i+1}{m_i+1}$ for it is not an integer and thus cannot be used as a valid window length. But we can still increase its m_i value to be 3 with which the resulting task set is still schedulable. Therefore we can adopt the floating redundant job scheme on it. As such, $Y = \{\tau_3\}$.

The remaining two tasks cannot be applied with either the window transferring scheme (because their values of $m_i \frac{k_i+1}{m_i+1}$ are not integers) or the floating redundant job scheme (because the resulting task set is not schedulable). As a result, if we apply Algorithm 3 to determine their mandatory jobs, both of them will need to be partitioned using E-patterns based on their original (m, k)-constraints. Correspondingly, $Z = \{\tau_1, \tau_2\}$.

Assume all tasks arrive at time 0. With the mandatory jobs of tasks τ_1 and τ_2 determined based on E-pattern, under the same fault rate and idle power assumption as in the previous part, the estimated total energy consumption will be 149.90062 units, which is below the given energy budget constraint. As shown in Figure 5, in this case, without reclaiming the slack time from canceled periodic tasks the response time of the aperiodic task τ_5 will be 59 (with slack reclaiming the response time will be 40).

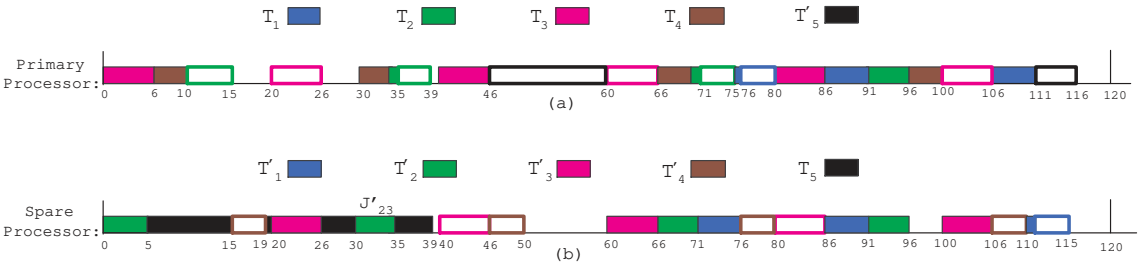


Fig. 6. (a) The schedule for the same periodic main tasks $\tau_1, \tau_2, \tau_3, \tau_4$ as in Figure 5(a) based on the hybrid pattern for the tasks in Z and the aperiodic backup task $\tau_5^{'}$ in the primary processor; (b) The schedule for the periodic backup tasks $\tau_1^{'}$, $\tau_2^{'}$, $\tau_3^{'}$, and $\tau_4^{'}$ based on the hybrid pattern and the aperiodic main task $\tau_5 = (19, 120)$ in the spare processor.

Algorithm 5 Modified task set partitioning using second round of Branch-and-Bound.

Input: task set $\tilde{T} = X \cup Y \cup Z$ output from Algorithm 3;
Output: task set $\hat{T} = X \cup Y \cup Z_R \cup Z_E$, where Z_R and Z_E are the subsets to be partitioned based on R-pattern and E-pattern, respectively;
$Z_R = \emptyset$;
Z_E = all tasks in Z;
$\hat{T} = X \cup Y \cup Z_R \cup Z_E$;
$U_{bound} = \sum_{\tau_i \in Z_R} \frac{m_i C_i}{k_i P_i}$;
Z-Partition (Z_R, Z_E, $\hat{T}$, U_{bound});
output ($\hat{T}$);

FUNCTION Z-Partition(Z_R, Z_E, $\hat{T}$, U_{bound})
for each task $\tau_i \in Z_E$ **do**
  $Z_R = Z_R \cup \{\tau_i\}$;
  Remove τ_i from Z_E;
  if $X \cup Y \cup Z_R \cup Z_E$ is schedulable **then**
    $Util = \sum_{\tau_i \in Z_R} \frac{m_i C_i}{k_i P_i}$;
    if $Util > U_{bound}$ **then**
      $U_{bound} = Util$;
      $\hat{T} = X \cup Y \cup Z_R \cup Z_E$;
    end if
    Z-Partition (Z_R, Z_E, $\hat{T}$, U_{bound});
  else
    put τ_i back to Z_E;
  end if
end for=0

However, if we adopt a different way of determining the mandatory jobs, *i.e*, letting τ_1 be partitioned based on a modified R-pattern in which optional jobs happen first while letting τ_2 be partitioned based on E-pattern, as shown in Figure 6, then the whole task set is still schedulable and, in this case, even without slack reclaiming the response time of the aperiodic task τ_5 will be reduced to 39 (with slack reclaiming the response time will be 30), which is much shorter than that in the above schedule

in Figure 5. Moreover, in this case the estimated total energy consumption is still 149.90062 units. Therefore the task set is still feasible.

As can be seen, there is great potential in minimizing the response time of the aperiodic tasks by adopting the (modified) R-pattern for the tasks in Z as the mandatory jobs in them can be "pushed back" at the maximal extent. However, since the schedulability of the R-pattern is not as good as E-pattern, it is possible that not all tasks can have their mandatory jobs determined in this way. For example, for the above task set, if the mandatory jobs of both τ_1 and τ_2 are determined based on (modified) R-pattern, the resulting task set will not be schedulable. Regarding that, the most reasonable way is to divide the sub task set Z further into two subsets Z_R and Z_E in which the tasks in Z_R have their mandatory jobs determined based on R-pattern while the tasks in Z_E have their mandatory jobs determined based on E-pattern. In order to do so, we can adopt a second round of branch-and-bound method on the task set $\tilde{T}$ output by Algorithm 3, which is sketched in Algorithm 5.

As shown in Algorithm 5, similar to Algorithm 3, our second round of branch-and-bound method determines task by task if each task $\tau_i \in Z$ should be partitioned based on R-pattern or E-pattern. When Algorithm 5 is finished, it is possible to reach certain hybrid configuration in which the tasks in subset Z_R are partitioned based on R-pattern while the tasks in subset Z_E are partitioned based on E-pattern. And the resulting configuration should be the one that can maximize the total utilization of the tasks in subset Z_R while guaranteeing the schedulability of the whole task set $\hat{T}$. Moreover, since for any task $\tau_i \in Z$, partitioning it based on R-pattern or E-pattern will generate the same number of mandatory jobs in it, the task set $\hat{T}$ output by Algorithm 5 will have the same estimated total energy consumption as the task set $\tilde{T}$ output by Algorithm 3. Therefore, the feasibility of the task set can still be guaranteed under the given energy constraint $\bar{E}_B$.

Note that after the sub task set Z was further divided into two parts, *i.e.*, Z_R and Z_E, the calculation of the delay period of φ_i in Equation (20) for each periodic task τ_i under the new hybrid job pattern should also be updated as follows.

$$\begin{aligned}\varphi_i = \min\{d_i \quad & - \sum_{D_x \le d_i}^{\tau_x \in X} (\lceil \frac{m_x}{y_x} \lceil \frac{d_i - D_x}{T_x} \rceil^+ \rceil)(C_x + t_{sc}) \\ & - \sum_{D_y \le d_i}^{\tau_y \in Y} (\lceil \frac{(m_y + 1)}{k_y} \lceil \frac{d_i - D_y}{T_y} \rceil^+ \rceil)(C_y + t_{sc}) \\ & - \sum_{D_u \le d_i}^{\tau_u \in Z_R} W_u^R(0, d_i) - \sum_{D_v \le d_i}^{\tau_v \in Z_E} W_v^E(0, d_i)\} \end{aligned} \tag{22}$$

Where

$$\begin{aligned} W_u^R(0, d_i) \quad = \quad & (m_u \lfloor \frac{\lceil \frac{d_i - D_u}{T_u} \rceil^+}{k_u} \rfloor \\ & + \quad \min\{(\lceil \frac{d_i - D_u}{T_u} \rceil^+ - m_u \lfloor \frac{\lceil \frac{d_i - D_u}{T_u} \rceil^+}{k_u} \rfloor), m_u\})(C_u + t_{sc}) \end{aligned}$$

is the mandatory work demand for task $\tau_u \in Z_R$ before d_i and

$$W_v^E(0, d_i) = (\lceil \frac{m_v}{k_v} \lceil \frac{d_i - D_v}{T_v} \rceil^+ \rceil)(C_v + t_{sc})$$

is the mandatory work demand for task $\tau_v \in Z_E$ before d_i .

Note that φ_i in Equation (22) can be used to delay any individual mandatory job of task τ_i. Since our goal is to minimize the response time(s) of the aperiodic task(s) in the middle priority band, we should try to delay the mandatory jobs of the periodic tasks in the upper priority band as late as possible. With φ_i, we can develop two sufficient conditions to help identify the maximal delay for the upcoming mandatory jobs in the upper priority band. The first one can be stated as followed (proof omitted).

THEOREM 4.1. *Assuming at time $t = t_0$, let M be the upcoming mandatory jobs with arrival times later than t_0. Let r_i be the arrival time for the next upcoming mandatory job of the periodic task τ_i. If the execution of M starts at*

$$T_{LS}(M) = \min(r_i + \varphi_i),\ i = 1, 2, \cdots, N \tag{23}$$

no mandatory job in M will miss its deadline.

The online complexity of computing $T_{LS}(M)$ in Theorem 4.1 is $O(N)$ since φ_i can be computed offline.

The second sufficient condition (similar to the one in [22]) can be stated as followed.

THEOREM 4.2. *Assuming at time $t = t_0$, let M be the set of upcoming mandatory jobs with arrival times later than t_0. Also let the delay bound (*i.e.*, the earliest deadline for the mandatory jobs in M) be T_{bd} for M. Then no mandatory job in M will miss its deadline if the execution of all mandatory jobs in M starts at $T_{LS}(M)$, where*

$$T_{LS}(M) = \min_{J_i \in J_s}(d_i^* - \sum_{J_k \in hp(J_i)} (C_k + t_{sc})), \tag{24}$$

where J_s consists of the jobs from M with arrival times earlier than T_{bd} but later than t_0, $hp(J_i)$ are the jobs with equal or higher priorities than J_i and

$$d_i^* = \min_p(d_i, r_p + \varphi_p), \forall J_p \in M, J_p \notin J_s \text{ and } d_p > d_i. \tag{25}$$

The main difference between Theorem 4.2 and the one in [22] is the way that the effective deadline d_i^* is defined. From equation (25), d_i^* is prolonged with the delay period φ_p of the mandatory jobs with arrive times later than T_{bd}. This in turn will allow the mandatory jobs to be delayed further.

The time complexity of computing $T_{LS}(M)$ in Theorem 4.2 is $O(N'M')$, where N' is the total number of mandatory jobs arriving within the interval from the arrival time of job J_i to its deadline and M' is the number of jobs with arrivals before T_{bd}. Since N' and M' are usually very small for periodic task sets, Theorem 4.2 provides a very efficient way to compute the delay for the upcoming mandatory jobs of the periodic tasks as well.

Note that since both Theorem 4.1 and Theorem 4.2 are sufficient conditions, the larger one from equation (23) and (24) should be used to delay the upcoming mandatory jobs for the periodic tasks to facilitate early completion of the aperiodic task(s). For example, in Figure 6 (b), the execution of job $J_{23}^{'}$ can be delayed to time $t = 35$, which will allow the response time of the aperiodic task τ_5 to be further reduced to 33.

5 EVALUATION

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

- *SSNEM* This is the naive approach in which the periodic tasks were partitioned with E-pattern, and the mandatory jobs in the primary and the spare processors were executed concurrently without delay.

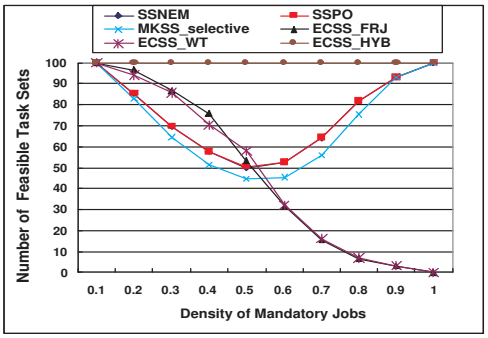


Fig. 7. Feasibility comparison of the different approaches.

- *SSPO* The periodic tasks were partitioned with E-pattern to satisfy the given (m, k)-constraints. Then the mandatory jobs were scheduled with the preference oriented scheme in [11] but without applying DVFS.
- $MKSS_{Selective}$ The periodic tasks were scheduled with the approach from [25] but adapted to EDF scheme. The periodic tasks were firstly partitioned with deeply-red pattern to satisfy the given (m, k)-constraints. Then the selective approach in [25] was applied.
- $ECSS_{FRJ}$ The periodic tasks were partitioned with E-pattern to satisfy the given (m, k)-constraints. Then the mandatory jobs were scheduled with the floating redundant job scheme proposed in Section 3.1.
- $ECSS_{WT}$ The periodic tasks were partitioned with E-pattern to satisfy the given (m, k)-constraints. Then the mandatory jobs were scheduled with the window transferring scheme proposed in Section 3.2.
- $ECSS_{HYB}$ The periodic tasks were partitioned with the hybrid pattern proposed in Section 4 to satisfy the given (m, k)-constraints. Then the mandatory jobs were scheduled with the modified approach proposed in Section 4.

For all the approaches compared, the aperiodic tasks were executed in the middle priority band with priority levels lower than the mandatory jobs but higher than the optional jobs from the periodic tasks. For the processor model we adopted a widely used due-core processor model, *i.e.*, the Samsung Exynos 4210 processor model [1]. According to [1], the highest speed that the Exynos 4210 processor core can operate is 1200MHz with power consumption of 1067.5 mWatt per core. We assumed the processor idle power $P_{idle} = 50$ mWatt and minimal shut-down interval $t_{sd} = 2$ millisecond. Meanwhile, the energy and time overheads for doing sanity (or consistency) checks, *i.e.*, E_{sc} and t_{sc} were assumed to be 0.2 mJoule and 0.1 millisecond, respectively.

5.1 Evaluation based on synthesized task sets

The task set tested in our experiments contains five to ten periodic tasks whose periods were randomly chosen in the range of $[10, 50]ms$. The deadlines of the periodic tasks were set to be less than or equal to their periods. The m_i and k_i for the (m, k)-constraints were also randomly generated such that k_i was uniformly distributed between 2 to 10, and $1 \leq m_i \leq k_i$. The worst case execution time (WCET) of a periodic task was uniformly distributed between 1 and its deadline. The task set can also contain some aperiodic task(s) whose worst case execution times was/were randomly chosen in

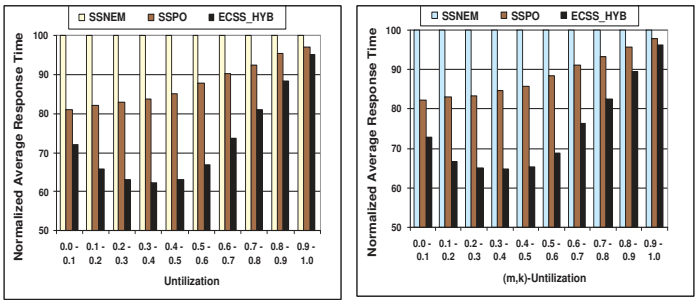
the range of $[10, 50]ms$. Each aperiodic task was assigned a soft deadline which is equal to the hyper period of the periodic tasks.

Firstly, we inspected the feasibility of the different approaches under different density of mandatory jobs of the periodic tasks. 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 periodic tasks under their original (m, k)-constraints and all aperiodic tasks 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 7. From Figure 7, 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 7, 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 55%, 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, 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 [23].

Next, we inspected the average response time of the aperiodic tasks by 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 the performance, we checked the average response time of the aperiodic tasks by the different approaches based on the utilization of the periodic tasks, *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.

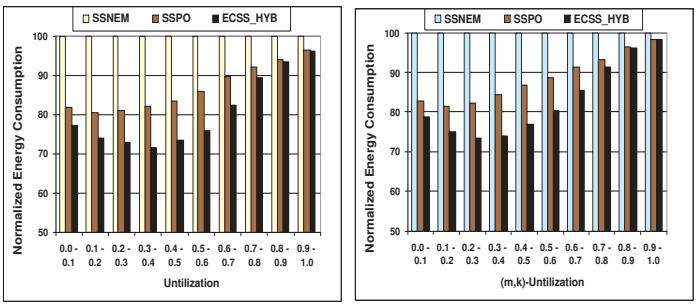
In the first set, we checked the average response time of the aperiodic tasks when no fault occurred during the hyper period. The results normalized to that by *SSNEM* are shown in Figure 8(a).

From Figure 8(a), it is easy to see that even when all approaches were feasible, our newly proposed approach, *i.e.*, $ECSS_{HYB}$ can reduce the average response time of the aperiodic tasks significantly



(a) (b)

Fig. 8. Average aperiodic response time for systems subject to (a) No faults; (b) System faults.



(a) (b)

Fig. 9. Energy consumption for systems subject to (a) No faults; (b) System faults.

compared with the previous approaches, *i.e.*, *SSNEM* and *SSPO*. The maximal reduction by $ECSS_{HYB}$ over *SSNEM* and *SSPO* can be up to 38% and 22%, respectively. The main reason is that, by adopting the hybrid approach in Section 4 and running the main job of the aperiodic tasks in the spare processor, $ECSS_{HYB}$ can help minimize the interference from the mandatory backup jobs of the periodic tasks (running in the higher priority band) on the aperiodic tasks (running in the middle priority band). Moreover, by delaying the execution of the mandatory backup jobs as late as possible with the sufficient conditions proposed in Section 4, the preemptions from the mandatory backup jobs of the periodic tasks on the aperiodic tasks can be greatly reduced, which is also quite helpful to the execution and early completion of the aperiodic tasks.

In the second set, we assumed the system could be subject to transient and/or permanent faults. The transient fault model is similar to that in [44] by assuming Poisson distribution with an average fault rate of 10^{-5} per millisecond. As for permanent fault, we assume it is distributed evenly along the time and at most one permanent fault will occur during the hyperperiod of the corresponding task set. The result is shown in Figure 8(b).

As could be seen, under this scenario, the average aperiodic response time achievable by our new approach, *i.e.*, $ECSS_{HYB}$ is still much less than the previous approaches. The maximal reduction by $ECSS_{HYB}$ over *SSNEM* and *SSPO* can be up to 35% and 20%, respectively, for the same reasons as stated above.

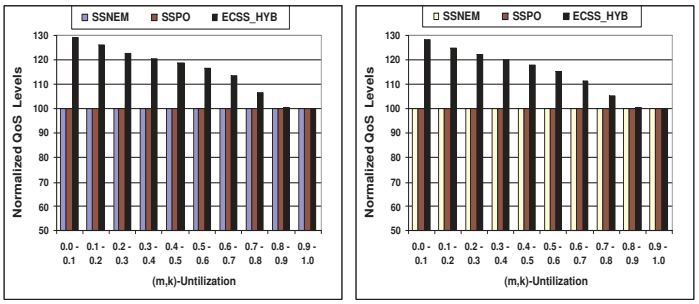


Fig. 10. QoS for systems subject to (a) No faults; (b) System faults.

Still next, we inspected the actual energy consumption of the different approaches. We also 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 9(a).

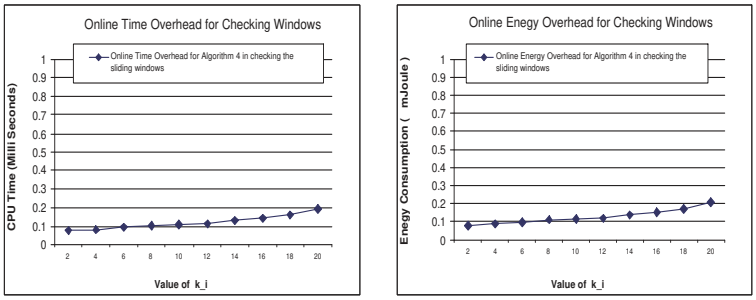
From Figure 9(a), it is easy to see that when all approaches were feasible, both the approaches with energy management, *i.e.*, $ECSS_{HYB}$ and *SSPO* still consumed much less actual energy than the approach without energy management, *i.e.*, *SSNEM*. Moreover, the actual energy consumption of $ECSS_{HYB}$ 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 18% less than that by *SSPO*. The main reason is that, under this scenario, by adopting the hybrid approach in Section 3.3, $ECSS_{HYB}$ can help minimize the overlapped execution between the mandatory jobs and their backup jobs of the periodic tasks 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 (22) also helped save energy consumption effectively.

In the second set, we assumed the system could be subject to permanent and/or transient faults with the same fault rate as in the previous group of test. The result is shown in Figure 9(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 16%. 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 faulty job using the delay intervals calculated in Equation (22) 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 number of effective jobs completed successfully 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 10(a). From Figure 10(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 nearly 30%. This is because, different from *SSNEM* and *SSPO* which could only provide a minimum set of



(a) (b)

Fig. 11. The online overhead of Algorithm 4 in checking the energy consumption of all sliding dynamic windows containing the current optional job: (a) Time overhead; (b) Energy overhead.

jobs that "just" satisfied the (m, k)-constraints, $ECSS_{HYB}$, by adopting hybrid configurations, could not only have extra number of mandatory jobs scheduled under the floating redundant job scheme and the window transferring scheme, but also dynamically scheduled some optional jobs for the tasks in Z while keeping the total energy consumption bounded under the given energy budget constraint. Therefore it could generally generate more valid jobs in its schedule, resulting in 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 10(b).

From Figure 10(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.

Finally, we also investigated the online overhead of Algorithm 4 in checking all the sliding dynamic windows containing the current optional job in terms of time and energy. In this part we varied the value of k_i from 2 to 20 and measured the accumulated CPU time for checking the sliding windows in it. Figure 11(a) illustrates the accumulated CPU time for the corresponding values of k_i. As shown in Figure 11(a), when the value of k_i is no larger than 20, the measured online time overhead is very small. The average value of it is slightly higher than 0.1 millisecond. Meanwhile, since in our approach there is no speed scaling on the processor, the online energy overhead also follows the same trend. As seen in Figure 11(b), in most cases the value of it is less than 0.2 mJoule when k_i is no larger than 20. In Algorithm 4 , this energy overhead has been incorporated into the estimation of the energy consumption for checking all the sliding dynamic windows containing the current optional job.

5.2 Evaluation based on real world benchmark

In this section, we tested our conclusions in a more practical environment.

The test is based on an real world benchmark: VCS (Vehicle Control System) [18]. The timing parameters such as the deadlines, periods, and execution times were adopted from the practical application directly [18]. The timing parameters of the aperiodic tasks were generated in the same way as in Section 5. The m_i and k_i values for the (m, k)-constraint were randomly generated between 2 and 10 ($k_i > m_i$).

We firstly performed two sets of experiments to inspect the aperiodic response time of the different approaches.

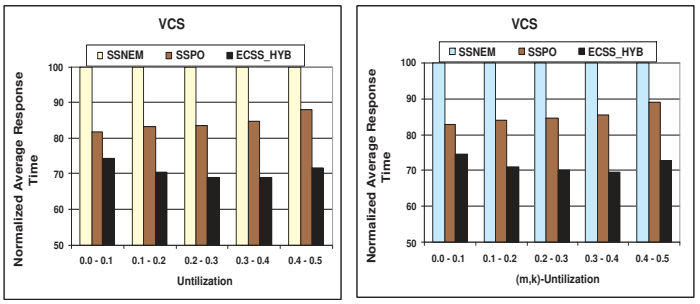


Fig. 12. Comparisons on the average aperiodic response time for systems subject to (a) No faults; (b) System faults.

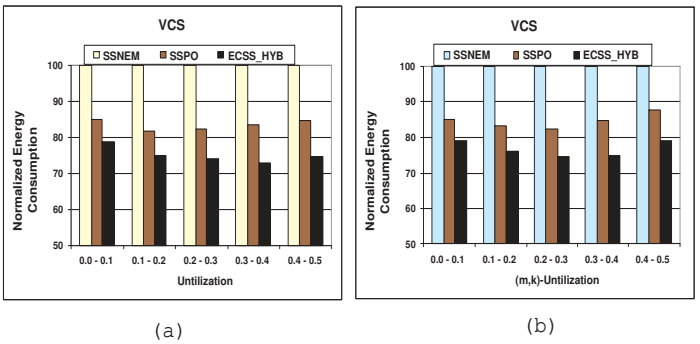


Fig. 13. Comparisons on the actual energy consumption for systems subject to (a) No faults; (b) System faults.

In the first set, we checked the average response time of the aperiodic tasks when no fault occurred within the hyperperiod. The results, normalized to that by *SSNEM*, are shown in Figure 12(a).

From Figure 12(a), it is easy to see that for VCS application, similar to the synthesized case, the average aperiodic response time of $ECSS_{HYB}$ is much less than all the other approaches for the same reasons as stated in Section 5. The maximal reduction by $ECSS_{HYB}$ over *SSNEM* and *SSPO* can be up to 27% and 15%, respectively.

In the second set, we assumed the system could be subject to permanent and/or transient faults with same fault rate as in Section 5. The result is shown in Figure 12(b).

As could be seen, under this scenario, the average aperiodic response time achievable by our new approach, *i.e.*, $ECSS_{HYB}$ is still much less than the previous approaches. The maximal reduction by $ECSS_{HYB}$ over *SSNEM* and *SSPO* can be up to 25% and 13%, respectively, for the same reasons as stated above.

Next, we inspected the actual energy consumption of the different approaches. We also conducted two sets of tests.

In the first set, we checked the energy performance of the approaches when no fault occurred during the hyper period. The results, normalized to *SSNEM* are shown in Figure 13(a).

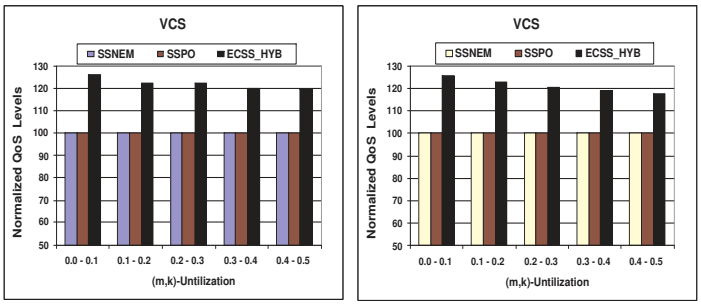


Fig. 14. Comparisons on the QoS for systems subject to (a) No faults; (b) System faults.

From Figure 13(a), it is easy to see that, similar to the results for randomly generated tasks, 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 the same reasons as stated in Section 5. When the system workload is not high, the actual energy consumed by $ECSS_{HYB}$ can be around 16% less than that by *SSPO*.

In the second set, we assumed the system could be subject to permanent and/or transient faults with the same fault rate as in the previous group of test. The result is shown in Figure 13(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 15% for the same reasons as stated above.

Finally, with the QoS in mind, we also inspected the QoS levels that the different approaches could provide when all approaches were feasible. 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 14(a). From Figure 14(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 28% mainly because $ECSS_{HYB}$ adopted hybrid configurations which could generate more valid jobs for the same reasons as stated for the synthesized task sets in Section 5.1.

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 14(b).

From Figure 14(b), the QoS improvement subject to faults by our newly proposed approach, *i.e.*, $ECSS_{HYB}$ over the previous approaches is still quite significant for the same reasons as stated above. Compared with *SSNEM* and *SSPO*, the maximal QoS improvement could be around 27%.

Overall, the evaluation results based on both synthesized systems and real world application have clearly demonstrated the effectiveness of our approaches in reducing average aperiodic response time as well as saving energy and improving QoS levels while satisfying the (m, k)-constraints and assuring fault tolerance through standby-sparing.

6 RELATED WORK

In last decades, plenty of work has been done in integrating QoS assurance into scheduling for real-time systems. For mixed-criticality systems, Gettings *et al.* [10] and Bruggen *et al.* [35] proposed

new approaches that can provide QoS-guarantee for low-criticality tasks. Moreover, for general fixed-priority weakly-hard real-time systems, schedulability analysis based on the Mixed Integer Linear Programming (MILP) formulation are provided in [33]. For mixed systems consisting of both periodic and aperiodic tasks, Buttazzo *et al.* [5] studied minimizing aperiodic response times in a firm real-time environment without considering energy consumption. With given energy budget constraint in mind, Alenawy *et al.* [2] proposed an approach to reduce the number of (m, k)-violations for weakly hard real-time systems.

Recently, with fault tolerance becoming an important concern for ubiquitous computing systems, a lot of works ([19, 41, 44]) have been presented in combining fault tolerant scheduling and energy management for real-time embedded systems. Many of them have utilized time redundancy, *i.e.*, to re-execute recovery jobs, whenever possible, to compensate the faulty jobs. Most of them have focused on dealing with transient faults only.

Besides transient faults, the system could be subject to permanent faults as well. More recently, to provide better system dependability, there has been increasing interest in adopting standby-sparing technique to deal with both permanent and transient faults simultaneously. With energy consumption in mind, in [9, 14], online power management schemes applying DVFS in the primary processor and DPM in the spare processor were studied. To better utilize the slack time in both processors, mixed scheduling schemes which adopt the combination of DVFS and DPM schemes in both the primary and spare processors were explored in [12]. For standby-sparing systems with mixed criticality, advanced energy management schemes were proposed in [32]. The biggest contribution in it was to set up a scheme to reduce energy through convex optimization in combination with power management heuristics based on joint DVFS and DPM schemes in both the primary and the spare processors. When considering the chip thermal effect, peak-power-aware standby-sparing techniques utilizing energy management schemes were presented in [3]. For real-time systems based on fixed-priority scheduling policies, standby-sparing schemes based on procrastination of the backup tasks were studied in [15]. In [4], more advanced fixed-priority standby-sparing techniques based on preference oriented scheduling schemes were explored. In[30], a scheme based on reverse preference-oriented priority assignment is proposed which is shown to be able to approach the energy performance of a theoretical lower bound when coupled with the dual-queue based delaying mechanism. For weakly hard real-time systems, in [25], an energy-aware approach was proposed to combine the standby-sparing technique and (m, k)-deadlines to achieve better energy efficiency for task sets partitioned based on deeply-red pattern [17]. However, as shown in [23], the schedulability of deeply-red pattern is weaker than that of the evenly distributed pattern used in this paper.

For multicore/multiprocessor systems, some works have also been conducted for real-time systems with fault tolerance capability. In [42], a framework is proposed to maximize the system availability by improving the mean time to failure (MTTF). In [7], Das *et al.* proposed an offline approach for mapping tasks onto processor cores to minimize energy consumption for all processor fault-scenarios. In [31], Safari *et al.* proposed a energy-aware solution for mixed-criticality multicore systems, which exploited task-replication to improve the QoS of low-criticality tasks in overrun situation while satisfying reliability requirements. The work in [12] described an implementation of standby-sparing through sharing the spare processor among multiple primary processors in multicore platforms to improve the overall energy efficiency using DVFS. In [29, 30], Roy *et al.* proposed standby-sparing schemes for reducing energy consumption on heterogeneous multicore platforms by applying DVFS. With thermal effect in mind, peak-power-aware reliability management scheme were presented in [3] to meet power and thermal constraints on the chip through distributing power density on the whole chip. In [20], a reactive triple modular redundancy (TMR) scheme was studied for tolerating both transient and permanent faults. Although TMR can avoid the potential problem of undetected faults in standby-sparing systems using sanity(or consistency) checking, since it needs to have at least

three copies of each real-time job scheduled among which at least two must be executed entirely (the third copy could be (partially) canceled depending on the results of the previous two copies), its vast energy consumption is a grave concern [20].

Also note that most of the above energy-aware approaches are focused on reducing the energy as much as possible. However, for systems that are driven by power supplies with limited energy budget constraint, the above best-effort approaches might not be able to ensure that the system could remain operational during a well-defined mission cycle. For systems with given fixed energy budget for its operation, Zhao *et al.* [40] proposed an approach to maximize the overall reliability of the systems subject to transient faults only. To the best of our knowledge, scheduling for an energy-constrained systems subject to both permanent and transient faults has not been studied yet. In this work, we assume the system is operating in an energy-constrained environment in which the energy consumption of the system must not exceed a given fixed budget. Based on it, we explore maximizing the feasibility and system performance for mixed systems containing both periodic and aperiodic tasks in a weakly hard real-time environment under fault tolerance requirement.

7 CONCLUSION

QoS, fault tolerance, and energy budget constraint are among the primary concerns for the design of real-time embedded systems. In this paper, we firstly presented several novel standby-sparing schemes for the periodic tasks which can ensure feasibility for the standby-sparing systems under tighter energy budget constraint than the traditional ones. Then based on them we proposed integrated approaches for both periodic and aperiodic tasks to minimize the aperiodic response time whilst achieving better energy and QoS performance under the given energy budget constraint. Through extensive evaluations, our results demonstrated that the proposed techniques significantly outperformed the existing state of the art approaches in terms of feasibility and system performance for mixed systems containing both periodic and aperiodic tasks in a weakly hard real-time environment while ensuring fault tolerance under the given energy budget constraint.

ACKNOWLEDGMENTS

This work is partly supported by the U.S. NSF under grants HRD 2135345, ECCS 2302651, CNS/SaTC 2039583, HRD 1828811, CMMI 2240407, the Kempe Foundation of Sweden, the National Natural Science Foundation of China (No. 62072085), 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] 2013. Energy analysis and prediction for applications on smartphones. *Journal of Systems Architecture* 59, 10, Part D (2013), 1375 – 1382.

[2] T. A. AlEnawy and H. Aydin. 2005. Energy-Constrained Scheduling for Weakly-Hard Real-Time Systems. *RTSS* (2005).

[3] M. Ansari, A. Yeganeh-Khaksar, S. Safari, and A. Ejlali. 2020. Peak-Power-Aware Energy Management for Periodic Real-Time Applications. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 4 (2020), 779–788.

[4] Rehana Begam, Qin Xia, Dakai Zhu, and Hakan Aydin. 2016. Preference-oriented Fixed-priority Scheduling for Periodic Real-time Tasks. *J. Syst. Archit.* 69, C (Sept. 2016), 1–14.

[5] G. C. Buttazzo and M. Caccamo. 1999. Minimizing aperiodic response times in a firm real-time environment. *IEEE Transactions on Software Engineering* 25, 1 (1999), 22–32.

[6] Houssine Chetto and Maryline Chetto. 1989. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transction On Software Engineering* 15 (1989).

[7] Anup Das, Akash Kumar, and Bharadwaj Veeravalli. 2014. Energy-aware Task Mapping and Scheduling for Reliable Embedded Computing Systems. *ACM Trans. Embed. Comput. Syst.* 13, 2s, Article 72 (Jan. 2014), 27 pages.

[8] Dongkun Shin and Jihong Kim. 2006. Dynamic voltage scaling of mixed task sets in priority-driven systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 3 (2006), 438–453.

[9] A. Ejlali, B. M. Al-Hashimi, and P. Eles. 2012. Low-Energy Standby-Sparing for Hard Real-Time Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31, 3 (March 2012), 329–342.

[10] Oliver Gettings, Sophie Quinton, and Robert I. Davis. 2015. Mixed Criticality Systems with Weakly-hard Constraints. In *Proceedings of the 23rd International Conference on Real Time and Networks Systems* (Lille, France) *(RTNS '15)*. 237–246.

[11] Yifeng Guo, Hang Su, Dakai Zhu, and Hakan Aydin. 2015. Preference-oriented real-time scheduling and its application in fault-tolerant systems. *Journal of Systems Architecture* 61 (01 2015).

[12] Yifeng Guo, Dakai Zhu, Hakan Aydin, Jian-Jun Han, and Laurence Yang. 2017. Exploit Primary/Backup Mechanism for Energy Efficiency in Dependable Real-Time Systems. *Journal of Systems Architecture* 78 (06 2017).

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

[14] M. A. Haque, H. Aydin, and D. Zhu. 2011. Energy-aware Standby-Sparing Technique for periodic real-time applications. In *ICCD*.

[15] Mohammad A. Haque, Hakan Aydin, and Dakai Zhu. 2015. Energy-aware standby-sparing for fixed-priority real-time task sets. *Sustainable Computing: Informatics and Systems* 6 (2015), 81 – 93.

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

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

[18] J. Li, YeQiong Song, and F. Simonot-Lion. 2006. Providing Real-Time Applications With Graceful Degradation of QoS and Fault Tolerance According to (m,k)-Firm Model. *Industrial Informatics, IEEE Transactions on* 2, 2 (May 2006), 112–119. https://doi.org/10.1109/TII.2006.875511

[19] Zheng Li, Shangping Ren, and Gang Quan. 2015. Energy Minimization for Reliability-guaranteed Real-time Applications Using DVFS and Checkpointing Techniques. *Journal of Systems Architecture* 61, 2 (Feb. 2015), 71–81.

[20] FatemehSadat Mireshghallah, Mohammad Bakhshalipour, Mohammad Sadrosadati, and Hamid Sarbazi-Azad. 2019. Energy-Efficient Permanent Fault Tolerance in Hard Real-Time Systems. *IEEE Trans. Comput.* 68, 10 (2019), 1539–1545.

[21] Linwei Niu. 2011. Energy Efficient Scheduling for Real-Time Systems with QoS Guarantee. *Journal of Real-Time Systems* 47, 2 (2011), 75–108.

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

[23] Linwei Niu and Gang Quan. 2006. Energy Minimization for Real-time Systems With (m,k)-Guarantee. *IEEE Trans. on VLSI, Special Section on Hardware/Software Codesign and System Synthesis* (July 2006), 717–729.

[24] Linwei Niu and Gang Quan. 2015. Peripheral-conscious energy-efficient scheduling for weakly hard real¨Ctime systems. *International Journal of Embedded Systems* 7, 1 (2015), 11–25.

[25] Linwei Niu and Dakai Zhu. 2020. Reliable and Energy-Aware Fixed-Priority (m,k)-Deadlines Enforcement with Standby-Sparing. *DATE* (2020).

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

[27] G. Quan and X.(Sharon) Hu. 2000. Enhanced Fixed-Priority Scheduling with (m,k)-Firm Guarantee. In *RTSS*. 79–88.

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

[29] Abhishek Roy, Hakan Aydin, and Dakai Zhu. 2017. Energy-aware standby-sparing on heterogeneous multicore systems. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6.

[30] Abhishek Roy, Hakan Aydin, and Dakai Zhu. 2021. Energy-aware primary/backup scheduling of periodic real-time tasks on heterogeneous multicore systems. *Sustainable Computing: Informatics and Systems* 29 (2021), 100474. https://doi.org/10.1016/j.suscom.2020.100474

[31] Sepideh Safari, Mohsen Ansari, Ghazal Ershadi, and Shaahin Hessabi. 2019. On the Scheduling of Energy-Aware Fault-Tolerant Mixed-Criticality Multicore Systems with Service Guarantee Exploration. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2338–2354.

[32] S. Safari, S. Hessabi, and G. Ershadi. 2020. LESS-MICS: A Low Energy Standby-Sparing Scheme for Mixed-Criticality Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020), 1–1.

[33] Youcheng Sun and Marco Di Natale. 2017. Weakly Hard Schedulability Analysis for Fixed Priority Scheduling of Periodic Real-Time Tasks. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 171 (Sept. 2017), 19 pages. https://doi.org/10.1145/3126497

[34] A. Taherin, M. Salehi, and A. Ejlali. 2018. Reliability-Aware Energy Management in Mixed-Criticality Systems. *IEEE Transactions on Sustainable Computing* 3, 3 (July 2018), 195–208.

[35] G. v. d. Bruggen, K. Chen, W. Huang, and J. Chen. 2016. Systems with Dynamic Real-Time Guarantees in Uncertain and Faulty Execution Environments. In *2016 IEEE Real-Time Systems Symposium (RTSS)*. 303–314.

[36] Yi wen Zhang. 2019. Energy-aware mixed partitioning scheduling in standby-sparing systems. *Computer Standards and Interfaces* 61 (2019), 129 – 136.

[37] Yi wen Zhang, Hui zhen Zhang, and Cheng Wang. 2017. Reliability-aware low energy scheduling in real time systems with shared resources. *Microprocessors and Microsystems* 52 (2017), 312 – 324.

[38] Richard West, Yuting Zhang, Karsten Schwan, and Christian Poellabauer. 2004. Dynamic Window-Constrained Scheduling of Real-Time Streams in Media Servers. *IEEE Trans. on Computers* 53, 6 (June 2004), 744–759.

[39] Ying Zhang, K. Chakrabarty, and V. Swaminathan. 2003. Energy-aware fault tolerance in fixed-priority real-time embedded systems. In *Computer Aided Design, 2003. ICCAD-2003. International Conference on*. 209–213.

[40] Baoxian Zhao, Hakan Aydin, and Dakai Zhu. 2010. On Maximizing Reliability of Real-Time Embedded Applications under Hard Energy Constraint. *IEEE Trans. Industrial Informatics* (2010), 316–328.

[41] Baoxian Zhao, Hakan Aydin, and Dakai Zhu. 2012. Energy Management Under General Task-Level Reliability Constraints. In *Proceedings of the 2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium (RTAS '12)*. Washington, DC, USA, 285–294.

[42] Junlong Zhou, Xiaobo Sharon Hu, Yue Ma, Jin Sun, Tongquan Wei, and Shiyan Hu. 2019. Improving Availability of Multicore Real-Time Systems Suffering Both Permanent and Transient Faults. *IEEE Trans. Comput.* 68, 12 (2019), 1785–1801.

[43] Dakai Zhu. 2011. Reliability-aware dynamic energy management in dependable embedded real-time systems. *ACM Trans. Embed. Comput. Syst.* 10 (January 2011), 26:1–26:27. Issue 2.

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