Energy Management for Fault-Tolerant (m,k)-Constrained Real-Time Systems that Use Standby-Sparing

LINWEI NIU, Howard University, Washington, USA
DANDA B. RAWAT, Howard University, Washington, USA
DAKAI ZHU, University of Texas at San Antonio, San Antonio, USA
JONATHAN MUSSELWHITE, Howard University, Washington, USA
ZONGHUA GU, Umeå University, Umeå, Sweden
QINGXU DENG, Northeastern University, Shenyang, China

Fault tolerance, energy management, and quality of service (QoS) are essential aspects for the design of real-time embedded systems. In this work, we focus on exploring methods that can simultaneously address the above three critical issues under standby-sparing. The standby-sparing mechanism adopts a dual-processor architecture in which each processor plays the role of the backup for the other one dynamically. In this way it can provide fault tolerance subject to both permanent and transient faults. Due to its duplicate executions of the real-time jobs/tasks, the energy consumption of a standby-sparing system could be quite high. With the purpose of reducing energy under standby-sparing, we proposed three novel scheduling schemes: the first one is for $(1, 1)$-constrained tasks, and the second one and the third one (which can be combined into an integrated approach to maximize the overall energy reduction) are for general (m, k)-constrained tasks which require that among any k consecutive jobs of a task no more than $(k - m)$ out of them could miss their deadlines. Through extensive evaluations and performance analysis, our results demonstrate that compared with the existing research, the proposed techniques can reduce energy by up to 11% for $(1, 1)$-constrained tasks and 25% for general (m, k)-constrained tasks while assuring (m, k)-constraints and fault tolerance as well as providing better user perceived QoS levels under standby-sparing.

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

Additional Key Words and Phrases: energy efficiency, fault tolerance, standby-sparing, QoS, real-time systems

1 INTRODUCTION

With the advance of IC technology, energy conservation has been a critical design issue for real-time embedded systems. With energy efficiency in mind, a lot of techniques have been proposed to reduce energy from different abstract levels (e.g. [1–3], *etc*). Among them the system level energy management has been a widely adopted approach. On the other hand, fault tolerance has also been a major concern for pervasive computing systems as system fault(s) could occur anytime [4]. Generally, computing system faults can be classified into permanent faults

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; Dakai Zhu, University of Texas at San Antonio, 1 UTSA Circle, San Antonio, TX, USA, 78249, dakai.zhu@utsa.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.

ACM 1539-9087/2024/2-ART
https://doi.org/10.1145/3648365

and transient faults [5]. 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 [5].

Recently a lot of research studies (e.g. [6, 7]) have been conducted on improving the energy efficiency for fault-tolerant real-time systems. Many of them have focused on dealing with transient faults. A widely adopted strategy is to utilize 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 are especially hazardous and need to be dealt with by all means to ensure system safety. Otherwise unrecoverable system failure could occur. More recently, solutions adopting hardware redundancy are proposed to address this issue. Among them the *standby-sparing* technique has gained much attention [8–10]. 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 [10]. 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 time constraint the executions of the main jobs in the primary processor and their corresponding backup jobs in the spare processor might need to be overlapped with each other. Thus the total energy consumption could be quite considerable. Regarding that, some recent works have been reported to reduce energy (e.g. [8–10]). The main idea is to try to let the executions of the main jobs and their corresponding backup jobs be shifted away as much as possible such that, once the main jobs are completed successfully, their corresponding backup jobs could be canceled early, thereby saving energy. With that in mind, in [9, 10], approaches based on EDL (earliest deadline as late as possible) scheme [11] were proposed for standby-sparing real-time systems. Their works are mainly focused on *hard* real-time systems, *i.e.*, the systems which require all real-time tasks/jobs meet their deadlines. However, in practical time-sensitive applications, such as multimedia or time-critical systems (for example, Webphone [12] and Vehicle Control System [13]), 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 data cannot be successfully reconstructed [14].

The *weakly hard real-time model* is more appropriate to model such kind of systems. Under the weakly hard real-time 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) [15]. Two well known weakly hard models are the (m, k)-model [16] and the *window-constrained* model [17]. The (m, k)-model requires that no more than $(k - m)$ jobs out of any *sliding* window of k consecutive jobs of the task could miss their deadlines, whereas the *window-constrained* model (represented as m/k-model) requires that no more than $(k - m)$ jobs out of each *fixed* and *nonoverlapped* window of k consecutive jobs could miss 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.* [18] adopted a partitioning strategy which divides the jobs into *mandatory* and *optional* ones. The mandatory ones are the jobs that must meet their deadlines in order to satisfy the (m,k)-constraints. In other words, so long as all the mandatory jobs can meet their deadlines, the (m,k)-constraints can be ensured. In [17], West *et al.* tried to set up a correspondence relationship between the *window-constrained* model and the (m,k) model. They found that the *window-constraints* can be converted to the (m,k)-constraints through certain automatic way.

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

- We proposed an efficient scheduling scheme for reducing energy consumption for $(1,1)$-constrained [19] tasks under standby-sparing.
- We further proposed two flexible and adaptive standby-sparing techniques which could be combined to maximize the overall energy reduction for general (m,k)-constrained tasks based on mandatory/optional job partitioning strategy.

The rest of the paper is organized as follows. In Section 2 we discuss the related work. Section 3 presents the preliminaries. Section 4 presents our approach for $(1,1)$-constrained tasks. Sections 5, 6, and 7 present our approaches for general (m,k)-constrained tasks. In Section 8 and Section 9, we present our evaluation results and conclusions.

2 RELATED WORK

In last decades, many research studies have been done in integrating QoS assurance into scheduling for real-time systems. For systems with transient overloaded conditions, Chetto *et al.* [20] explored scheduling algorithms for firm real-time systems. For mixed-criticality systems, Gettings *et al.* [21] and Bruggen *et al.* [22] 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 was provided in [23]. Considering given energy budget constraint, Alenawy *et al.* [24] proposed an approach to reduce the number of (m,k)-violations for weakly hard real-time systems. Also to minimize the number of dynamic failures, Kooti *et al.* [25] proposed a QoS-aware approach for (m,k)-firm real-time systems with long-term variations of the harvested energy.

Recently, with fault tolerance becoming an important concern for ubiquitous computing systems, a lot of works (for example, [4, 7, 26–28]) 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.

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 [8, 9, 29, 30], 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 [31]. 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 [33, 34]. Their approach targeted minimizing the peak-power of the standby-sparing systems such that the total power consumption generated by the chip would not exceed what the cooling component was designed to dissipate under any workload. Most of the above works are for real-time systems based on dynamic priority

scheduling policies. For real-time systems based on fixed-priority scheduling policies, standby-sparing schemes based on procrastination of the backup tasks were studied in [35, 36]. In [37, 38], more advanced fixed-priority standby-sparing techniques based on task level preference oriented scheduling schemes were explored. As shown in later part of this paper, such kind of task level preference oriented schemes could procrastinate some jobs unnecessarily, which might not be most energy efficient from the system point of view.

For multicore/multiprocessor systems, some works have also been conducted for real-time systems with fault tolerance capability. In [39], a framework is proposed to maximize the system availability by improving the mean time to failure (MTTF). In [40], Das *et al.* proposed an offline approach for mapping tasks onto processor cores to minimize energy consumption for all processor fault-scenarios. In [41], Safari *et al.* proposed an 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 [31] 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 [38, 42, 43], standby-sparing schemes applying DVFS for reducing energy consumption based on heterogeneous multicore platforms were proposed. With thermal effect in mind, thermal-aware power/reliability management scheme were presented in [33, 44, 45] to meet power and thermal constraints on the chip through distributing power density on the whole chip. In [46], 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 [46, 47].

Note that all of the aforementioned existing fault-tolerant schemes have focused on *hard* real-time systems only. For weakly hard real-time systems, in [48], an energy-aware scheme 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 [49]. However, as shown in [14], the schedulability of deeply-red pattern is generally weaker than that of the evenly distributed pattern used in this paper. Moreover, the approaches in [48] must rely on the execution of optional jobs, which cannot deal with the tasks with $(1, 1)$-constraint in which no optional jobs are available. The novelty of our proposed work in this paper lies in the fact that we are dealing with task set partitioned based on evenly distributed pattern. Moreover, our approaches do not always have to rely on the execution of optional jobs. So it can deal with the tasks with $(1, 1)$-constraint as well.

3 PRELIMINARIES

3.1 System models

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

The system consists of two identical processors which are denoted as primary processor and spare processor, respectively. For the purpose of tolerating permanent/transient faults, each mandatory job of a task τ_i has two duplicate copies running in the primary and the spare processors separately. Whenever a permanent fault is encountered in either processor, the other one will take over the whole system (to continue as normal). For

convenience, we call each task τ_i *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.

3.2 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 [50] as

$$P_{dyn} = \alpha C_{eff} V_{dd}^2 f \tag{1}$$

where α is the switching activity, C_{eff} is the effective switching capacitance, V_{dd} is the supply voltage, and f is the system clock frequency. The static power (P_{st}) can be expressed as

$$P_{st} = L_g(V_{dd} I_{subn} + \mid V_{bs} \mid I_j) \tag{2}$$

where L_g is the number of devices in the CMOS circuit , I_{subn} is the subthreshold leakage current, V_{bs} is the body bias voltage, and I_j is the reverse bias junction current in the 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}) which is equal to P_{st} whose major portion comes from the leakage. 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 DVFS techniques, the efficiency of DVFS in reducing the overall energy is becoming seriously degraded with the dramatic increase in leakage power (as part of the static power) with the shrinking of IC technology size [51]. With that in mind, in this paper we assume that the processors and the hardware platform used for standby-sparing do not apply DVFS. As such, when the processor is busy, it always consumes P_{busy} at the maximal speed s_{max}. Moreover, since Dynamic power down (DPD), *i.e.*, put the processor into its sleeping state, can greatly reduce the leakage energy when the processor is not in use, we assume that when no job is pending for execution, the processors can be put into sleeping state with DPD. But DPD needs to consume energy/time overheads for implementing shutting-down/waking-up the processor dynamically. If we assume the energy overhead and time overhead of DPD to be E_o and t_o, respectively, the processor can be shut down with positive energy gains when the length of the idle interval is larger than $t_{sd} = \max(\frac{E_o}{P_{idle} - P_{sleep}}, t_o)$. Correspondingly we call t_{sd} the *minimal shut-down interval*.

3.3 Fault Model

Similar to the standby-sparing systems in [9, 10], 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 [52] and the overhead for detection can be integrated into the job's execution time. Moreover, following the fault model in [4], we assume that the transient faults will present Poisson distribution [53] 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, for any

job J_{ij} of task τ_i, the reliability of it, represented by $\gamma(J_{ij})$, is defined as the probability that J_{ij} could be completed successfully under the maximal speed s_{max}. According to [4], $\gamma(J_{ij})$ is given as:

$$\gamma(J_{ij}) = e^{-\sigma(s_{max})c_{ij}} \tag{4}$$

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

3.4 Problem Formulation

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

PROBLEM 1. *Given system $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_N\}$, schedule $\mathcal{T}$ with EDF scheme in a standby-sparing system such that the total energy consumption can be minimized while the (m, k)-constraints for all tasks could be satisfied under the fault tolerant requirement.*

4 STANDBY-SPARING FOR $(1, 1)$-CONSTRAINED TASK SETS

For task sets with $(1, 1)$-constraint (or task sets in which $m_i = k_i$ for all tasks), under standby sparing, all jobs need to have two duplicate copies running in the primary and the spare processors, respectively. It is not hard to see that, due to the overlapped executions between them, one way to save energy is to let each main job in the primary processor be executed as soon as possible and its backup job in the spare processor be executed as late as possible such that, once the main job is completed successfully, the remaining part of its backup job can be canceled immediately, therefore saving part of the energy for executing the backup job. To this end, in [9] Mohammad *et. al* proposed to run the main tasks in the primary processor according to the EDF scheme and the backup tasks on the spare processor according to the earliest deadline as late as possible (EDL) scheme such that the overlapped executions between the main jobs and their backup jobs could be reduced, enabling energy savings. The energy reduction could be further boosted by adopting the preference oriented earliest deadline scheduling scheme in [10]. The main idea of the preference oriented scheme in [10] is to assume the whole task set could be divided into two disjoint subsets, *i.e.*, the subset to be executed as soon as possible (ASAP subset) and the subset to be executed as late as possible (ALAP subset). For any real-time tasks to be scheduled, if it belongs to the ASAP subset in one processor, then its backup task should belong to the ALAP subset in the other processor, and *vice versa*. Their idea could be demonstrated using the following example:

Given a task set of three tasks, *i.e.*, $\tau_1 = (4, 16, 20, 1, 1)$, $\tau_2 = (6, 17, 20, 1, 1)$, $\tau_3 = (6, 38, 40, 1, 1)$, to be executed in a standby-sparing system. While it is not stated in [10] how to divide the task set into ASAP subset and ALAP subset, here for convenience we assume a straightforward method of dividing the tasks, *i.e.*, letting the tasks τ_1 and τ_2 belong to the ASAP set and the task τ_3 belong to the ALAP set. Moreover, since the QoS constraints for the tasks are all $(1, 1)$-constraints which equals to the periodic hard real-time case in which all jobs are "mandatory", there is no optional job in these tasks. For simplicity, in this section all jobs refer to the mandatory jobs.

By applying the preference oriented scheme in [10], the main tasks τ_1 and τ_2 and the backup task $\tau_3^{'}$ will be scheduled in the primary processor (with τ_1 and τ_2 executed as soon as possible while $\tau_3^{'}$ executed as late as possible) while backup tasks $\tau_1^{'}$ and $\tau_2^{'}$ and main task τ_3 will be scheduled in the spare processor (with $\tau_1^{'}$ and $\tau_2^{'}$ executed as late as possible while τ_3 executed as soon as possible). If we assume no fault occurred, the complete schedules for them within the hyperperiod $[0, 40]$ is shown in Figure 1(a) and (b), respectively. As a result, the total active energy consumption within the hyperperiod is 32 units [1].

[1]For easy of presentation, in all examples in this paper we normalize P_{busy} (under the maximal processor speed 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.

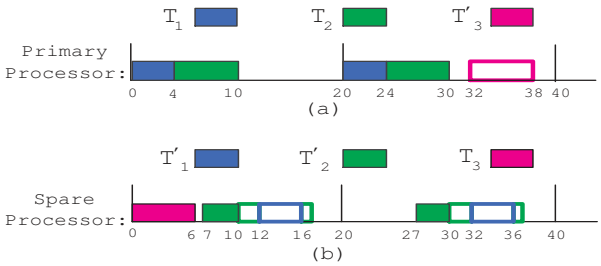


Fig. 1. (Emptied block(s) represent(s) the slack time generated by the canceled part(s) of the job(s).) (a) The schedule for the main tasks τ_1 and τ_2 as well as backup task $\tau_3^{'}$ in the primary processor under the preference oriented scheme [10]; (b) The schedule for the backup tasks $\tau_1^{'}$, $\tau_2^{'}$ as well as main task τ_3 in the spare processor under the preference oriented scheme [10].

Fig. 2. Assuming some task missed deadline at time point t',

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

THEOREM 4.1. *Given a task set $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_N\}$, if the release time(s) of any job(s) under the EDF schedule is/are procrastinated to its/their corresponding delayed starting time(s)[2] under the EDL schedule, all task deadlines can be guaranteed.*

PROOF. We use contradiction. Assuming under the EDF schedule, after the release time(s) of some job(s) are delayed to their starting times (*i.e.*, the delayed release times) under the EDL scheule, at certain time point t', some task missed its deadline. Then as shown in Figure 2, we can always find another time point $t_0 < t'$ such that during the time interval $[t_0, t']$ the processor is kept busy executing only jobs with release times or delayed release times no earlier than t_0 and with deadlines less than or equal to $(t' - t_0)$. Since no job has release time earlier than time 0, t_0 is well defined. Then the total work demand within the interval $[t_0, t']$ is bounded by $\sum_{D_q \le (t'-t_0)} \lceil \frac{t'-t_0-D_q}{T_q} \rceil^+ C_q$. Since some job missed the deadline at t', we have

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

On the other hand, consider the scenario when we delay the release times of all jobs within the interval $[t_0, t']$ to their starting times under the EDL schedule. Then in this case there must be a time point t_1 ($t_0 \le t_1 < t'$, as shown in Figure 2) such that during the interval $[t_0, t_1]$ the processor is either idle or executing jobs with deadlines larger

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

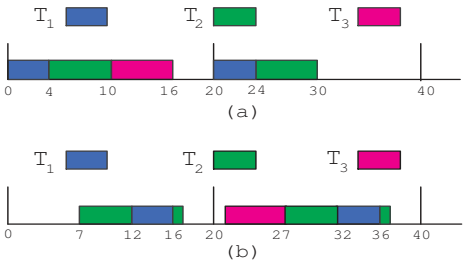


Fig. 3. The schedule for the task set under: (a) the EDF scheme; (b) the EDL scheme.

than $(t' - t_0)$ while during the interval $[t_1, t']$ the processor is busy executing only the jobs with deadlines less than or equal to $(t' - t_0)$. Moreover, the total work demand within $[t_1, t']$ will be no larger than the total work demand within the interval $[t_1, t']$ under the EDL schedule, *i.e.*,

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

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

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

Therefore, we have

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

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

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

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

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

Which Contradicts to (5)! □

To help understand Theorem 4.1, for the same task set in Figure 1, its schedules under EDF and EDL schemes are shown in Figure 3 (a) and (b), respectively. It is easy to verify that if the release time(s) of any job(s) (for example, J_{21}) in Figure 3 (a) is/are procrastinated to the delayed release time(s) of the same job(s) in Figure 3 (b), (for example, $\hat{r}_{21} = 7$ for task J_{21}), all deadlines in Figure 3 (a) can be guaranteed.

COROLLARY 4.2. *For an job J_i, its delayed starting time calculated by Theorem 4.1 is the maximal time it can be delayed to while ensuring the deadlines of all other tasks/jobs in the system.*

PROOF. The result follows the optimality of EDL in terms of procrastinating all jobs in the system safely while ensuring their deadlines. □

With Theorem 4.1, our new approach for scheduling the task set under standby-sparing can be implemented based on the **adaptive delay of individual jobs**, which is shown in Figure 4. As seen in Figure 4, unlike the preference oriented approach in Figure 1 which always (and only) delay all backup jobs (in the primary or the spare processor) uniformly, our approach will adaptively delay each individual job (either a main job or its corresponding backup job, but not both) depending on the actual need, which is summarized into the following *adaptive delay policies*:

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

Based on the above adaptive delay policies, as seen in Figure 4(a), at time $t = 0$, both job J_{11} and its backup job $J_{11}^{'}$ got a chance to be dispatched. However, since only one of them can be executed as soon as possible and the other one must be delayed, we just randomly picked one out of them, say J_{11}, to be executed as soon as possible in the primary processor (Figure 4(a)) while delaying its backup job $J_{11}^{'}$ in the spare processor to time $\hat{r}_{11} = 12$ (Figure 4(b)) (when J_{11} is completed successfully at time $t = 4$, the remaining time budget of $J_{11}^{'}$ will be inserted into the slack queue $\mathcal{S}$ of the spare processor). At the same time, since J_{21} will be preempted by J_{11} (due to its lower priority) and could not get a chance to be dispatched/executed at time $t = 0$ in the primary processor (Figure 4(a)) while its backup job $J_{21}^{'}$ will get a chance to be dispatched/executed at time $t = 0$ in the spare processor (Figure 4(b)), $J_{21}^{'}$ will be executed as soon as possible in the spare processor while J_{21} should be delayed to time $\hat{r}_{21} = 7$ in the primary processor (when $J_{21}^{'}$ is completed successfully at time $t = 6$, the remaining time budget of J_{21} will be inserted into the slack queue $\mathcal{S}$ of the primary processor). Similarly, at time $t = 4$, since J_{31} got a chance to be dispatched/executed first, it will be executed as soon as possible while its backup job $J_{31}^{'}$ will be delayed to time $\max\{\hat{r}_{31}, t + 4\} = 21$ (where 4 is the slack time with priority higher than or equal to $J_{31}^{'}$ in the spare processor) and executed as late as possible. Following the same rationale, the complete schedules within the hyperperiod [0,40] are shown in Figure 4(a) and (b). Under the same fault free assumption as in Figure 1, the total active energy consumption within the hyperperiod is reduced to 26 units, which is 19% lower than that in Figure 1.

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

As shown in Algorithm 1, each job (either a main or backup job) J_i has a category field associated with it whose value could be "E" (representing as early as possible execution) or "D" (representing as late as possible delay). Upon dispatching, if J_i got chance to be executed earlier than its corresponding job in the other processor, J_i's category should be set as "E" which means it should always be executed as early as possible (for example, J_{31} in Figure 4(a)). Otherwise J_i's category should be set as "D" which means it should always be delayed as late as possible (for example, $J_{31}^{'}$ in Figure 4(b)). Whenever a job is completed successfully, its corresponding job in the other processor should be canceled and the remaining part of its time budget will become slack time (line 17).

Note that, during run-time, in both the primary and the spare processors, a slack queue $\mathcal{S}$ needs to be maintained to keep track of the slack time(s) from (partially) canceled job(s). The slack time(s) in $\mathcal{S}$ will be sorted according to their deadline(s). Upon job completion, new slack time from the canceled job, if any, will be inserted into the slack queue $\mathcal{S}$ based on its deadline. Upon the dispatching of a job J_i at time t, the slack time from $\mathcal{S}$ with priorities

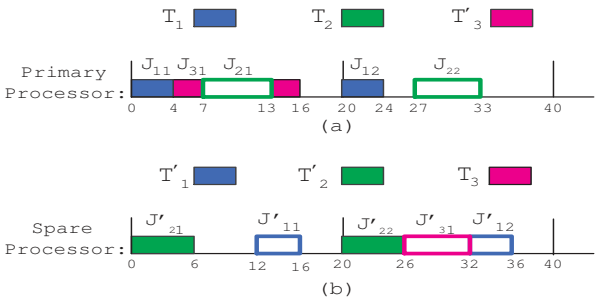


Fig. 4. The schedule for the main/backup jobs of each task under adaptive delay of individual job(s) in: (a) the primary processor; (b) the spare processor.

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

For either the primary processor or the spare processor:

Upon the execution of a job J_i at current time t_{cur}:
if its category is "E" **then**
  Execute it following the EDF scheme as soon as possible;
  if any slack time $S_i(t)$ with higher priority than J_i is available **then**
    Reclaim the slack time to execute J_i as soon as possible;
  end if
else
  // J_i's category is "D" and should be executed as late as possible;
  Revise the arrival time of J_i to $\max\{\hat{r}_i, (t_{cur} + S_i(t_{cur}))\}$;
  Execute J_i following the EDF scheme;
end if

Upon the completion of a job J_i at current time t_{cur}:
if the execution of job J_i is successful **then**
  Cancel its corresponding (backup) job in the other processor and add the residue time budget to the slack queue $\mathcal{S}$;
  if J_i was the only job in the job ready queue at time t_{cur}^- **then**
    Let NTA be the earliest (revised) arrival time of the next upcoming jobs(s) of all tasks;
    if $(NTA - t_{cur}) > t_{sd}$ **then**
      Shut down the processor and set wake-up timer as $(NTA - t_{cur})$;
    end if
  end if
end if

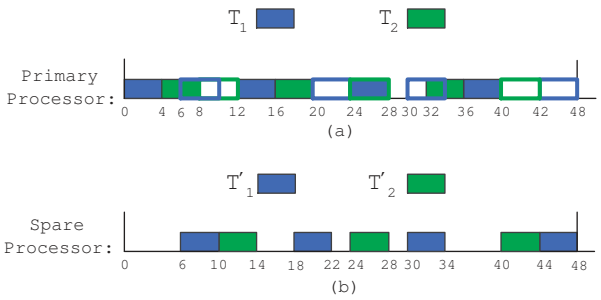


Fig. 5. (Emptied block(s) represent(s) the slack time generated by the canceled part(s) of the mandatory main/backup job(s).) (a) The schedule for the optional jobs (and canceled mandatory main jobs) for task set $\tau_1 = (4, 6, 6, 4, 8)$, $\tau_2 = (4, 8, 8, 2, 4)$ in the primary processor under the selective scheme in [48] based on E-pattern; (b) The schedule for the optional jobs (reclaiming the slack time from the canceled backup jobs) for the same task set in the spare processor under the selective scheme in [48] based on E-pattern.

higher than or equal to J_i will be stored in a variable $S_i(t)$. If J_i's category is "E", $S_i(t)$ should be reclaimed to execute J_i as soon as possible (line 6-8). Otherwise if J_i's category is "D", $S_i(t)$ should be used to implement dynamic procrastination on J_i, which can delay J_i to $\max\{\hat{r}_i, (t_{cur} + S_i(t))\}$ (line 11). Moreover, when the system is idle (or shut down), slack times in $\mathcal{S}$ should also be consumed based on their sorted sequence in $\mathcal{S}$.

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

5 STANDBY-SPARING FOR GENERAL (m, k)-CONSTRAINED TASK SETS BASED ON WINDOW-TRANSFERRING

For tasks with general (m, k)-constraint in which $m_i < k_i$, to ensure the (m, k)-deadlines, a widely adopted strategy is to judiciously partition the jobs into *mandatory* jobs and *optional* jobs [54]. Two well-known partitioning strategies are the *evenly distributed pattern* (or E-pattern) [18] and the *deeply-red pattern* (or R-pattern) [49]. 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):

$$\pi_{ij} = \begin{cases} \text{“1”} & \text{if } j = \lfloor \lceil \frac{(j-1)\times m_i}{k_i} \rceil \times \frac{k_i}{m_i} \rfloor + 1 \\ \text{“0”} & \text{otherwise} \end{cases} \qquad j = 1, 2, 3, \cdots \tag{11}$$

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

$$\pi_{ij} = \begin{cases} \text{“1”} & \text{if } 1 \le j \bmod k_i \le m_i \\ \text{“0”} & \text{otherwise} \end{cases} \qquad j = 1, 2, 3, \cdots \tag{12}$$

The mandatory/optional job partitioning according to equation (11) has the property that it helps to spread out the mandatory jobs *evenly* in each task along the time. Moreover, it is shown in [14] that E-pattern has better schedulability than R-pattern in general and is the optimal pattern when all task periods are co-prime in particular. In [14], a variation of E-pattern called E^R-pattern was achieved by reversing the pattern horizontally to let the optional jobs happen first, which can preserve the schedulability of E-pattern [14].

For task sets based on R-pattern, Niu *et al.*[48] proposed an approach by exploring the flexibility of executing jobs under (m, k)-deadlines to avoid executing duplicate copies of the mandatary jobs on two processors whenever possible. Their approach is based on selectively executing some optional jobs with flexibility degree of 1 (*i.e.*, the optional jobs right before the mandatory jobs) and, once they are completed successfully, their next mandatory job(s) will become optional (their backup jobs can be simply dropped to save energy) and this procedure can be progressed further. Their approach is generally efficient in saving energy for task sets already schedulable with R-pattern. However, as we mentioned, since the schedulability of R-pattern is weaker than E(or E^R)-pattern, there still exist a large number of task sets schedulable with E(or E^R)-pattern but not schedulable with R-pattern. For such kind of task sets, since there are much more optional jobs with flexibility degree of 1 in them, if we apply the approach in [48] to them, it could result in executing excessive number of optional jobs, which could adversely affect the overall energy efficiency. This could be demonstrated with the following example.

Consider another task set of two tasks, *i.e.*, $\tau_1 = (4, 6, 6, 4, 8)$, $\tau_2 = (4, 8, 8, 2, 4)$. It is easy to verify that this task set is schedulable under E(or E^R)-pattern but not schedulable under R-pattern. The job patterns based on E^R-pattern for them are "01010101", and "0101", respectively. As can be seen, due to their even distribution property, all optional jobs in them have a flexibility degree of 1. If we apply the approach in [48] to the task set, the schedules in the primary and the spare processors are shown in Figure 5(a) and (b), respectively. As shown in Figure 5(a), since all optional jobs (including those jobs demoted from mandatory to optional) in it have a flexibility degree [48] of 1, all of them will be selected for execution in the primary or the spare processor alternatively. If we assume no fault occurred within the first hyperperiod $[0, 48]$, all mandatory main/backup jobs in either the primary or the spare processor could be demoted/dropped. As a result the total active energy consumption within the hyperperiod is 56 units.

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

LEMMA 5.1. *[17] For any task τ_i with (m, k)-constraint of (m_i, k_i), if it can satisfy the* window-constraint *of $m_i/\frac{(m_i+k_i)}{2}$, its original (m, k)-constraint will be satisfied automatically.*

The above lemma provides us more opportunities to reduce the energy consumption under standby-sparing. Before presenting our new approach, we need to define a variation of the E-pattern as followed. Based on it, the pattern π_{ij} for job J_{ij}, is defined as [54]:

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

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 (11) 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 can determine the mandatory jobs of each main task τ_i and their backup jobs based on the window constraint of $m_i/\frac{(m_i+k_i)}{2}$ first. For the tasks in the same task set as in Figure 5, their corresponding window constraints will be $4/6$ and $2/3$, respectively. Then under E-pattern, the mandatory jobs of task τ_1 will be scheduled in the primary processor, as shown in Figure 6(a) and the backup jobs of τ_1 will be determined based on the E^{r_i}-pattern with $r_i = 1$ within each separate window of length $\frac{(m_i+k_i)}{2}$ and scheduled in the spare processor, as shown in Figure 6(b). Meanwhile, to balance the mandatory workload of two processors, we let the mandatory main jobs for task τ_2 based on E-pattern be scheduled in the spare processor and the backup jobs for them based on the E^{r_i}-pattern with $r_i = 1$ be scheduled in the primary processor, as shown in Figure 6. As such,

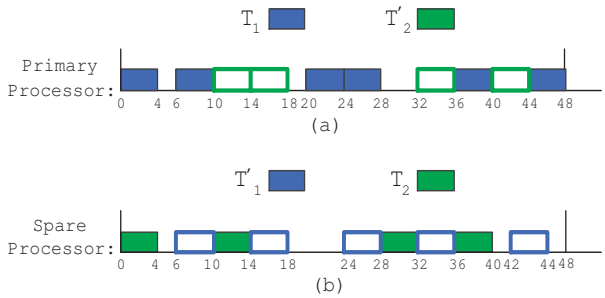


Fig. 6. The schedule for the mandatory main jobs based on E-pattern and (canceled) backup jobs based on E^{r_i}-pattern for same task set as in Figure 5 using our approach in: (a) the primary processor; (b) the spare processor.

each mandatory main job and its backup job will be shifted away completely such that once a mandatory main job is completed successfully, its backup job in the other processor could be canceled entirely. If any mandatory main job of task τ_i failed, its corresponding backup job in the other processor could still be invoked timely. In this way, the window-constraint of task τ_i could be guaranteed. Then according to Lemma 5.1, its original (m, k)-constraint can also be ensured. Following the same rationale, if we assume no fault occurred, the complete schedule within the hyperperiod is shown in Figure 6. The total active energy consumption for it is reduced to 40 units, which is 28.6% lower than that in Figure 5.

5.1 Dealing with transient faults

In the window-transferring approach above, 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 7, assuming job J_{11} in the primary processor has failed, then its backup job $\tilde{J}_{11}$ (*i.e.*, J'_{12} in this case) in the spare processor needs to be executed. Meanwhile, in the primary processor, the mandatory main job J_{12} will be executed in the same time frame as J'_{12}. Suppose the current job J_{12} is completed successfully. Under this scenario, if we had canceled its backup job $\tilde{J}_{12}$ (*i.e.*, J'_{13} in this case) in the spare processor, then in the time interval $[6, 12]$ there would be only one valid job because J_{12} and J'_{12} are in the same time frame and would effectively contribute only one valid job. Consequently the window constraint of $4/6$ will be violated in the time interval of [0, 36]. As a result its original (m, k)-constraint might not be ensured by Lemma 5.1.

The main reason for the above issue is that, due to the pattern rotation on the backup jobs, 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_{12} and J'_{12} in Figure 7). 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 be decreased by 1, which could cause the window-constraint in it to be violated and consequently the original (m, k)-constraint to be violated as well. Therefore, in this case, even if the current mandatory main job is completed successfully, its backup job should not be canceled.

Due to the above reason, when determining the backup job patterns, to save energy, we should let the backup job patterns be rotated in a way that the total number of mandatory jobs with their time frames overlapped (between

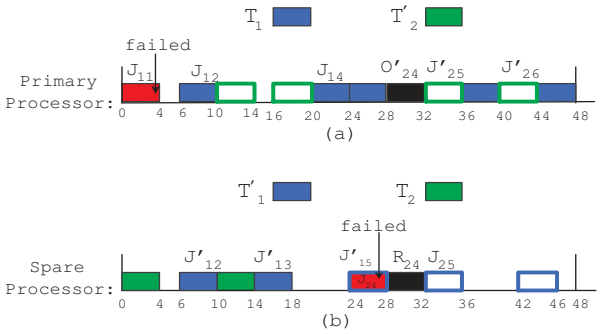


Fig. 7. (a) The schedule for the optional jobs (and canceled backup jobs) for task set $\tau_1 = (4, 6, 6, 4, 8)$, $\tau_2 = (4, 8, 8, 2, 4)$ in the primary processor under the window transferring scheme based on E-pattern; (b) The schedule for the optional jobs (and canceled backup jobs) for the same task set in the spare processor under the window transferring scheme based on E-pattern.

the mandatory main jobs and the backup jobs) is minimized. Fortunately, the backup job patterns determined based on the E^{r_i}-pattern with $r_i = 1$ has this property, which is summarized into the following lemma.

LEMMA 5.2. *For any task τ_i with (m, k)-constraint of (m_i, k_i), if $y_i = \frac{(m_i+k_i)}{2}$ is an integer, let the mandatory main jobs and the backup jobs of τ_i within each separate window of y_i jobs be partitioned based on E-pattern and E^{r_i}-pattern, respectively according to the new (m, k)-constraint of (m_i, y_i). Then the total number of overlapped mandatory jobs in time (represented as N_i) between the mandatory main jobs and the backup jobs is minimized when $r_i = 1$. Moreover, in this case $N_i = \max(0, 2m_i - y_i)$.*

PROOF. We will show that when $r_i = 1$, the total number of overlapped mandatory jobs in time is minimized by considering three cases in general: (i) $m_i = \frac{y_i}{2}$; (ii) $m_i < \frac{y_i}{2}$; (iii) $m_i > \frac{y_i}{2}$.

For case (i): when $m_i = \frac{y_i}{2}$, due to the property of even distribution of mandatory/optional jobs, the job pattern under E-pattern can only be in the form of 10... while the job pattern under the E^{r_i}-pattern can only be in the form of 01.... Obviously the total number of overlapped jobs between the mandatory main jobs and the (mandatory) backup jobs is 0 in this case, *i.e.*, $N_i = 0$. Therefore it is minimized.

For case (ii): when $m_i < \frac{y_i}{2}$, due to the property of even distribution of the mandatory/optional jobs, there is no consecutive mandatory jobs under E-pattern. In other words, each mandatory job is surrounded by optional jobs(s) only in both sides. So when we rotate the E-pattern to the right by one position to achieve the E^{r_i}-pattern with $r_i = 1$, each mandatory backup job in the E^{r_i}-pattern will correspond to an optional job in the original E-pattern, which means the total number of overlapped jobs between the mandatory main jobs and the (mandatory) backup jobs must be 0 in this case, *i.e.*, $N_i = 0$. Therefor it is also minimized.

For case (iii): when $m_i > \frac{y_i}{2}$, due to the property of even distribution of mandatory/optional jobs, there is no consecutive optional jobs under E-pattern or E^{r_i}-pattern. In other words, each optional job in the E-pattern or E^{r_i}-pattern is surrounded by mandatory jobs(s) only in both sides. So when we determine the backup job patterns by rotating the E-pattern to the right by one position to achieve the E^{r_i}-pattern with $r_i = 1$, each optional job in the E-pattern will correspond to a mandatory job in the E^{r_i}-pattern, and *vice versa*. Since in this case all optional jobs in the E-pattern or E^{r_i}-pattern have been "consumed" in terms of corresponding to the mandatory jobs in the other type of patterns, the total number of overlapped jobs between the mandatory main jobs and the (mandatory) backup

jobs must have been minimized. Moreover, in this case, since the total number of mandatory jobs from both the E-pattern and E^{r_i}-pattern are $2m_i$, we shall have

$$2N_i + (y_i - m_i) + (y_i - m_i) = 2m_i \tag{14}$$

Therefore,

$$N_i = 2m_i - y_i \tag{15}$$

From all the three cases above, when $r_i = 1$, N_i is minimized and $N_i = \max(0, 2m_i - y_i)$. □

As shown above, when $m_i > \frac{y_i}{2}$, $N_i = (2m_i - y_i) > 0$, which means the overlapping between the time frames of some mandatory main job(s) and the backup job(s) of some other mandatory main job(s) is inevitable. In this case, if the current mandatory main job overlapped with the backup job of some other failed mandatory main job, its backup job cannot be canceled even if it is completed successfully, which will consequently incur extra energy consumption. Regarding that, to save energy, if we could compensate the failed job in a different way, it is still possible to avoid the above consequence and therefore achieve better energy efficiency. This could be demonstrated as follows:

For the same task set in Figure 7, at time $t = 24$, since the backup job $J_{15}^{'}$ in the spare processor is canceled due to the successful execution of its mandatory main job J_{14} in the primary processor, the mandatory main job J_{24} can reclaim the slack time from $J_{15}^{'}$ for execution. Thereafter, suppose J_{24} is found to have failed at time $t = 26$. Then, instead of executing its backup job $J_{25}^{'}$ in the other processor, we could reserve a recovery job R_{24} for J_{24} at time $t = 26$ by reclaiming the remaining slack time from $S_i(t)$ at $t = 26$, which is 4 units and long enough to accommodate the recovery job R_{24} in this case. Once reserved, the recovery job R_{24} will be scheduled like a mandatory job. If R_{24} is completed successfully, we can still cancel the backup job of J_{24}, *i.e.*, $J_{25}^{'}$, as shown in Figure 7(b). Thus we can avoid having the next mandatory main job J_{25} overlapped with the backup job of J_{24}, *i.e.*, $J_{25}^{'}$. So the situation in the above case (*i.e.*, for jobs J_{12} and $J_{12}^{'}$) can be avoided.

Note that it is also possible that the available slack time $S_i(t)$ is not enough to accommodate a recovery job for the failed mandatory main job. In this case there is still another option to compensate the failure of the mandatory main job. That is, we can check the other processor to see whether it is possible to run the optional job from the backup task that corresponds to the failed mandatory main job in time. If the optional job could be completed timely and successfully, it will be counted as a valid job which could compensate the failed mandatory main job as well. For example, for job J_{24} in this case, if $S_i(t) < 4$, we can still consider invoking the optional job $O_{24}^{'}$ in the other processor at time $t = 26$. If it can be completed successfully, it has the same effect as reserving and executing a recovery job for J_{24}. However, whether to invoke the optional job or not needs to be handled carefully because it has lower priority than the mandatory jobs and could potentially be preempted by other mandatory and/or optional jobs and caused to miss its deadline. Obviously, if the optional job cannot meet the deadline, it has no benefit to energy saving at all. Regarding that, an optional job should be executed, non-preemptively by any other optional job, only when it could be guaranteed to be finished by the earliest arrival time of the nearest upcoming mandatory job in the same processor.

5.2 Reliability analysis

In this section, we will analyze the performance of the proposed window-transferring scheme in terms of reliability.

For any task τ_i, the reliability of an arbitrary window of k_i consecutive jobs in it is defined as its probability of having at least m_i jobs out of it completed successfully.

It is not hard to see that, for any task τ_i with $y_i = \frac{(m_i+k_i)}{2}$ being an integer, since its separate window length $y_i \leq k_i$, if we inspect each sliding window of k_i jobs in it, it is possible that in some sliding window of k_i jobs there are some redundant job(s) in it, *i.e.*, the total number of mandatory jobs in it could exceed m_i. For example,

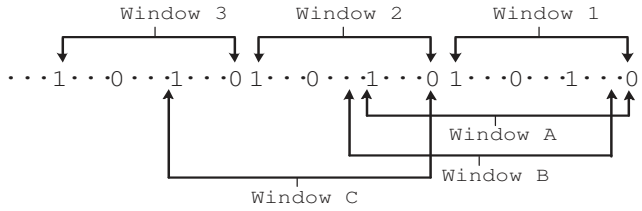


Fig. 8. The job patterns after window transferring.

in the sample task set in Figure 4, the original (m, k)-constraint of task τ_1 is $(4, 8)$. However, as shown in Figure 5, since after window transferring, τ_1's mandatory jobs was determined based on its window-constraint of $4/6$. The corresponding job pattern for τ_1 becomes "110110110110110110 $\cdots$ " in which each sliding window of 8 jobs contains at least one redundant job in it. Generally speaking, for any task τ_i with y_i being an integer, after window transferring based on Lemma 1, with high probability there will be at least one redundant job within any arbitrary window of k_i jobs in it, which is summarized into the following lemma.

LEMMA 5.3. *For any task τ_i with (m, k)-constraint of (m_i, k_i) $(0 < m_i < k_i)$, let the mandatory jobs of τ_i within each separate window of $y_i = \frac{(m_i+k_i)}{2}$ jobs be partitioned based on E-pattern according to the new (m, k)-constraint of (m_i, y_i). The probability for τ_i to have at least one redundant job in any sliding window of k_i jobs in it is 1 if $1 < m_i < (y_i - 1)$ and $(1 - \frac{1}{y_i})$ otherwise.*

PROOF. Since $\tau_i \in \mathcal{A}$, we know $y_i = \frac{(m_i+k_i)}{2}$ is an integer. Since in the general case, for all (m, k)-constraint in which $m_i < k_i$, we have $m_i < y_i$ as well. So the value of m_i can only be $1, \cdots, (y_i - 1)$. We consider three cases in general, (i) $1 < m_i < (y_i - 1)$; (ii) $m_i = 1$; (iii) $m_i = (y_i - 1)$.

For case (i), without lose of generality, let's assume the case when two consecutive windows of y_i jobs at the rightmost of the hyperperiod, namely window 1 and window 2, partitioned with E-pattern are shown in Figure 8. Considering the rightmost sliding window of k_i jobs in the current position of Window A, obviously there are at least m_i "1"s in it because $k_i \geq y_i$ (thus Window 1 is contained in Window A completely). Moreover, Since $k_i = y_i + (y_i - m_i)$, Window A also includes exactly $(y_i - m_i)$ job patterns in Window 2. Obviously, $(y_i - m_i)$ equals the number of '0's in Window 2. Since $1 < m_i < (y_i - 1)$, due to the even distribution property of E-pattern, any $(y_i - m_i)$ job patterns must contain at least a '1' in it. Therefore the number of '1's in Window A is no less than $(m_i + 1)$. Moreover, each time when we move window A to the left by one position, due to the property of E-pattern, before it reaches the position of Window C, the number of '1's in it will not decrease. Once it reaches the position of Window C, the situation in the beginning will repeat. Therefore we can conclude that in this case all sliding windows of y_i jobs much contain at least $(m_i + 1)$ '1's, *i.e.*, at least one redundant mandatory job in it.

For case (ii), the situation in the beginning is similar to case (i), the only difference is that since $m_i = 1$, all the $(y_i - m_i)$ job patterns in the right part of Window 1 (which are covered by Window A as well) are '0's. So the number of '1's in Window A is exactly m_i. But when it moves to the left by one job position, the number of '1's in it will be increased by one. If we move Window A to the left further, the number of '1's in it will not change until it reach the position of Window C, where the situation in the beginning will repeat. Therefore in this case, among every y_i sliding window of k_i consecutive jobs, there are exactly one out of them containing m_i '1's in it while in all of the other cases there are at least $(m_i + 1)$ '1's in them. In other words, the chance for any window of k_i consecutive jobs to contain no redundant mandatory job is $\frac{1}{y_i}$

For case (iii), the situation is similar to case (ii), only in the beginning there are exactly m_i '1's in window A. All the other sliding window of k_i jobs before Window C has at least $(m_i + 1)$ '1's in them. Once the sliding window

reaches the position of Window C, the situation in the beginning will repeat. So, the chance for any window of k_i consecutive jobs to contain no redundant mandatory job is also $\frac{1}{y_i}$.

□

Note that, for each individual mandatory job, the probability for it to be completed successfully is $\gamma(J_{ij})$ defined in Equation (4). Therefore its probability of encountering transient fault is $(1 - \gamma(J_{ij}))$. Moreover, since under standby-sparing, each mandatory job has a backup job, a mandatory job is regarded as having failed (and counted as a job failure within its window) only when both the mandatory main job and its backup job have failed, whose probability is

$$(1 - \gamma(J_{ij}))^2 \tag{16}$$

Also the mandatory job is counted as a valid job if either the mandatory main job is completed successfully or the mandatory main job has failed but its backup job is completed successfully, whose probability is

$$\gamma(J_{ij}) + (1 - \gamma(J_{ij})) \times \gamma(J_{ij}) \tag{17}$$

As such, for any sliding window of k_i jobs in task τ_i, based on Lemma 5.3, if $1 < m_i < (y_i - 1)$, since the total number of mandatory jobs in it is $(m_i + 1)$ (because there must be a redundant job in it), its probability of having one mandatory job failed with m_i mandatory jobs completed successfully out of it is

$$\binom{1}{m_i + 1} \times (1 - \gamma(J_{ij}))^2 \times [\gamma(J_{ij}) + (1 - \gamma(J_{ij})) \times \gamma(J_{ij})]^{m_i} \tag{18}$$

Also its probability of having $(m_i + 1)$ mandatory jobs out of it completed successfully is

$$[\gamma(J_{ij}) + (1 - \gamma(J_{ij})) \times \gamma(J_{ij})]^{m_i+1} \tag{19}$$

Therefore under this scenario its window-level reliability, *i.e.*, the probability of having at least m_i jobs out of it completed successfully, represented as $\gamma(W_i)$, should be the summation of the above two cases, *i.e.*, $\gamma(W_i) = (18) + (19)$.

Otherwise (*i.e.*, $1 < m_i < (y_i - 1)$ is not true, under which $m_i = 1$ or $(y_i - 1) \leq m_i \leq y_i$), for any sliding window of k_i jobs in task τ_i, according to Lemma 5.3, its probability of containing no redundant job and at least one redundant job in it is $\frac{1}{y_i}$ and $(1 - \frac{1}{y_i})$, respectively. Therefore, following the same rationale, its window-level reliability $\gamma(W_i)$ should be calculated as

$$\begin{aligned}\gamma(W_i) \quad = \quad & \frac{1}{y_i}[\gamma(J_{ij}) + (1 - \gamma(J_{ij})) \times \gamma(J_{ij})]^{m_i} + (1 - \frac{1}{y_i}) \times \{\binom{1}{m_i + 1} \times (1 - \gamma(J_{ij}))^2 \times [\gamma(J_{ij}) \\ + \quad & (1 - \gamma(J_{ij})) \times \gamma(J_{ij})]^{m_i} + [\gamma(J_{ij}) + (1 - \gamma(J_{ij})) \times \gamma(J_{ij})]^{m_i+1}\}\end{aligned} \tag{20}$$

As we know, under the original E-pattern, for any task τ_i, since there are exactly m_i jobs out of any sliding window of k_i jobs [18], its window-level reliability $\gamma(W_i)$ should be calculated as

$$\gamma(W_i) = [\gamma(J_{ij}) + (1 - \gamma(J_{ij})) \times \gamma(J_{ij})]^{m_i} \tag{21}$$

For example, for the task τ_1 in Figure 5, if we assume the probability of transient fault under the maximal speed to be 10^{-5} per millisecond, its job level reliability according to Equation (4) for each mandatory job is $\gamma(J_{ij}) = e^{10^{-5} \times 4} = 0.99996$. Therefore before window transferring, since based on E-pattern any sliding window in it contains exactly m_i mandatory jobs, its window-level reliability based on Equation (21) is $(0.99996 + (1 - 0.99996) \times 0.99996)^4 = 0.9999999936$. Therefore, before window transferring, the probability for any window of k_i jobs of task τ_1 to encounter dynamic failure will be $(1 - 0.9999999936) = 6.4 \times 10^{-9}$

However, after window transferring, its window-level reliability based on the summation of Equations (18) and (19) is $\binom{1}{5} \times (1-0.99996)^2 \times [0.99996+(1-0.99996) \times 0.99996]^4 + [0.99996+(1-0.99996) \times 0.99996]^5 = 0.99999999957$.

Therefore, after window transferring, the probability for any window of k_i jobs of task τ_1 to encounter dynamic failure will be $(1 - 0.99999999957) = 4.3 \times 10^{-10}$. As seen, after window transferring, the probability for any window containing k_i jobs of task τ_1 to encounter dynamic failure is much lower (by nearly one order of magnitude less) than that before window transferring.

Based on the above calculation of the window-level reliability, the reliability of task τ_i should be calculated as,

$$\gamma(\tau_i) = \prod_{q=1}^{z_i} \gamma(W_{iq}) \tag{22}$$

where $\gamma(W_{iq})$ represents the reliability of the q^{th} window of task τ_i and z_i is the total number of windows inspected for task τ_i within the hyper period.

Then based on (22), the reliability of the whole system $\mathcal{T}$ should be calculated as,

$$\gamma(\mathcal{T}) = \prod_{i=1}^{n} \gamma(\tau_i) \tag{23}$$

More results on the reliability of the whole system and its probability to encounter dynamic failure under the general case can be found in Section 8.1.2

5.3 Probability of tolerating undetected faults

When we use sanity (or consistency) checks [52] to determine if the execution of the main mandatory job is successful or not, it is possible that such kind of tests are not 100% accurate [55]. Sometimes a fault may remain undetected [55, 56]. If such kind of situation occurs, dynamic failure will be inevitable in the existing approach based on the original (m, k)-pattern because any failure on any mandatory job in them will cause the (m, k)-constraint to be violated. In other words, their probability of tolerating such kind of checking errors is zero. On the contrary, our approach based on window transferring has much better capability of tolerating such kind of checking errors mainly due to the existence of the redundant mandatory jobs. From Lemma 5.3, the probability for a task τ_i in $\mathcal{A}$ to tolerate at least one undetected fault in each and any window of k_i jobs is 1 if $1 < m_i < (y_i - 1)$, and $(1 - \frac{1}{y_i})$ otherwise. More results on the general case can be found in Section 8.1.2.

6 STANDBY-SPARING FOR GENERAL (m, k)-CONSTRAINED TASK SETS BASED ON DYNAMIC PATTERN VARIATION

Although the above approach based on window-transferring has great potential in energy saving , it also has the problem that, since $\frac{(m_i+k_i)}{2} \leq k_i$, after the mandatory jobs are determined under window constraint based on Lemma 5.1, some task(s) might become non-schedulable. On the other hand, if we want to maintain the scheduability by using the original k_i value as the window length and adopt E-pattern to determine mandatory main jobs in the primary processor while adopting E^{r_i}-pattern to determine the backup jobs in the spare processor, it is possible that the (m, k)-constraint in some sliding window will be violated if some mandatory main job failed because E(or E^{r_i})-pattern only contains the minimal number of mandatory jobs that "just" satisfy the (m, k)-constraint. In this case the failed mandatory main job could not be compensated by its backup job timely due to the strictness of the original (m, k) requirement. To save energy for the tasks whiling maintaining the original schedulability, a more promising way is to cautiously execute some optional job(s) when possible and vary the patterns of the future jobs dynamically, which could be demonstrated with the following example.

Consider another task set of three tasks, *i.e.*, $\tau_1 = (2, 2.5, 5, 2, 4)$, $\tau_2 = (1, 3, 10, 3, 6)$, $\tau_3 = (3, 6, 15, 2, 4)$. If we apply the window-transferring scheme based on Lemma 5.1, τ_1 and τ_3 can have their mandatory main/backup jobs determined based on the window-constraints of $2/3$ and $2/3$, respectively. However, it is not hard to verify that after transferring the resulting task set became non-schedulable. So the above window-transferring scheme should

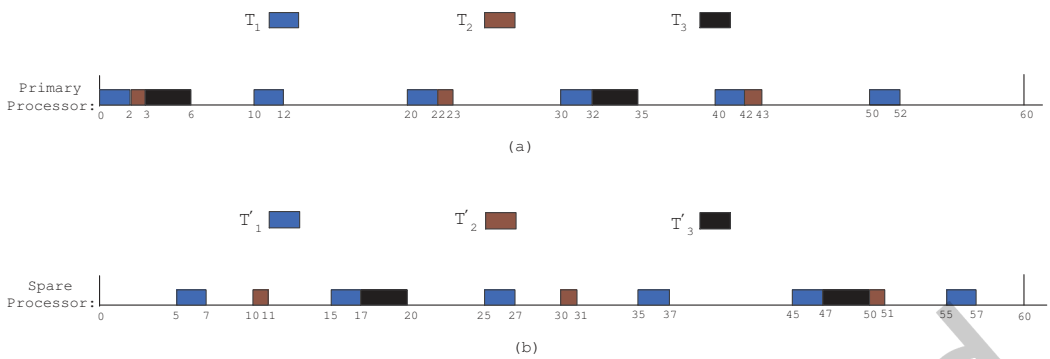


Fig. 9. The schedule for the task sets $\tau_1 = (2, 2, 5, 2, 4),\ \tau_2 = (1, 3, 10, 3, 6),\ \tau_3 = (3, 6, 15, 2, 4)$ based on E-pattern using the approach from [48] in: (a) the primary processor; (b) the spare processor.

not be adopted. On the other hand, it is also easy to verify that this task set is non-schedulable under R-pattern but only schedulable under E (or E^R)-pattern. So the approach in [48] cannot be applied to it based on R-pattern. Regarding that, if we partition the jobs of all tasks under the original (m, k)-constraint using E^R-pattern and apply the approach in [48] to it, the fault-free schedule within the hyperperiod $[0, 60]$ is shown in Figure 9 and the total active energy consumption for it will be 42 units. Note that in this schedule since all optional jobs were executed (because the flexibility degrees of all of them are 1), no optional job got chance to be skipped.

However, if we adopt a different approach in scheduling the optional/mandatory jobs, the energy efficiency could be improved further. As shown in Figure 10(a), at time point $t = 0$, the optional job J_{11} could be executed and completed timely. If the execution of J_{11} was successful, its next mandatory job J_{12} could be demoted to optional (the backup job of J_{12} in the spare processor could be dropped). Note that in order to ensure the original (m, k)-constraints, all future job patterns of task τ_1 could be varied by restarting the E^R-pattern from the next job position. Since the optional jobs do not need backup jobs for them, the execution of J_{11} should be quite helpful in reducing the energy for executing the mandatory main job J_{12} together with its backup job under the old patterns. Similarly, at time point $t = 3$, the optional job J_{31} could be executed and completed timely. If the execution of J_{31} was successful, its next mandatory job J_{32} could be demoted to optional and the E^R-pattern of task τ_3 will be restated from there. Then at the execution of time $t = 15$, ed job J_{32} could be re-executed as an optional job and further demoted J_{33} to optional. Similarly, J_{33} could also be re-executed as an optional job at time $t = 30$. It is easy to see that, without effective control, this procedure will be repeated further and the complete schedule will be very similar to that in Figure 9 except that in Figure 9 the optional jobs were executed in different processors. Obviously that will have no benefit in energy savings compared with Figure 9. Hence, in order to reduce energy, here we need to control the number of optional jobs to be executed. One reasonable way to implement that is to check, if the current optional job is executed, whether the expected window energy WE'_i of the current dynamic window which the optional job belongs to will exceed the original static window energy WE_i without running optional jobs. If $WE'_i > WE_i$, there is no benefit to run the optional job and it should be skipped; otherwise it is energy beneficial to run the optional job and it should be executed. For example, in Figure 10, the WE'_3 values for optional job J_{31}, J_{32}, and J_{33} are 3, 6, and 9, respectively which are smaller than the original window energy WE_3 for task τ_3 (after partial job cancellation, the value of WE_3 is estimated as 11). So J_{31}, J_{32}, and J_{33} will all be executed (and the job pattern will be varied correspondingly). However, the WE'_3 value for optional job J_{34} will be 12 which is larger than WE_3. So J_{34} should not be executed. Since J_{34} is skipped. no pattern variation on J_{35}. So J_{35} together with its backup job

still need to be executed as mandatory job. Following the same rationale, the complete schedule within the first hyperperiod $[0, 60]$ is shown in Figure 10 and the total active energy consumption for it is 33.5 units, which is 20% lower than that from the schedule in Figure 9

From the above example, we can see that, under E-pattern, the execution of optional jobs needs to be dealt with carefully. Selecting optional jobs for execution solely based on flexibility degree might not always be most energy efficient. Instead, incorporating the energy information into the selection criteria can enhance the energy efficiency significantly. Regarding that, we will formulate the optional job selection criteria into an algorithm as shown in Algorithm 2.

Algorithm 2 The execution/skip-over of optional jobs for task τ_i

Input: static window energy WE_i for task τ_i;
Output: The boolean value of executable function, *i.e.*, $Executable(J_{ij})$ for the current optional job J_{ij} of task τ_i;

For the primary processor:

Upon the arrival of optional job J_{ij} of task τ_i at current time t_{cur}:
if J_{ij} is an optional job **then**
  Executable (J_{ij}) = **true**;
  $WE'_i = c_{ij}$;
  for $y = 1$ to $(k_i - 1)$ **do**
    if $J_{i(j-y)}$ is an optional job **then**
      if $J_{i(j-y)}$ has been executed **then**
        $WE'_i += c_{i(j-y)}$;
      end if
    else
      $WE'_i += c_{i(j-y)} + ac'_{i(j-y)}$; // $ac'_{i(j-y)}$ is the actual executed part of backup job J'_{ij}
    end if
  end for
  if $WE'_i > WE_i$ **then**
    Executable (J_{ij}) = **false**;
  end if
end if

Note that in Algorithm 2, one of the essential parts is to estimate the static window energy WE_i for task τ_i that is needed as input for Algorithm 2. Since the exact amount of cancelled part in the backup jobs is unpredictable during the offline phase, here we will adopt an heuristics in estimating the value of WE_i. It is not hard to see that, under the original (m, k)-pattern, for any particular mandatory job J_{ij}, if we assume its backup job $\tilde{J}_{ij}$ (in this case $\tilde{J}_{ij}$ is the same as J'_{ij}) can be postponed all the way to its deadline D_i and executed continuously between the completion time of its main job and D_i, the expected overlapped execution time between J_{ij} and its backup job $\tilde{J}_{ij}$ will be $\max\{0, C_i - (D_i - \hat{R}_i)$, where $\hat{R}_i$ is the average response time of task τ_i. Correspondingly, the average energy consumed by J_{ij} together with its backup job (after possible cancellation) is $C_i + \max\{0, C_i - (D_i - \hat{R}_i)$. Therefore, the average energy consumption within any sliding window of k_i jobs will be

$$\Sigma_{j=1}^{k_i} \pi_{ij}(C_i + \max\{0, C_i - (D_i - \hat{R}_i)) = m_i(C_i + \max\{0, C_i - (D_i - \hat{R}_i)) \quad (24)$$

To limit the number of optional jobs selected for execution and skip over optional jobs as necessary, we will use the value calculated in the above Equation (24) as our static window energy WE_i for task τ_i, which will be used as the input for Algorithm 2.

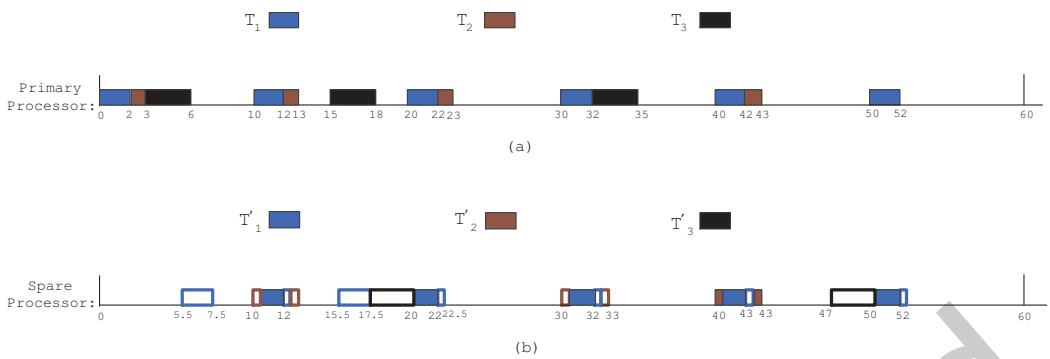


Fig. 10. The schedule for task sets $\tau_1 = (2, 2, 5, 2, 4)$, $\tau_2 = (1, 3, 10, 3, 6)$, $\tau_3 = (3, 6, 15, 2, 4)$ based on E-pattern using our dynamic pattern variation approach in: (a) the primary processor; (b) the spare processor (empty rectangles represent the cancelled part of the corresponding mandatory (or backup) jobs).

Note that in Equation (24), the value of $\hat{R}_i$, *i.e.*, the average response time of task τ_i could be calculated statistically through simulation.

7 COMBINED STANDBY-SPARING APPROACH FOR GENERAL (m, k)-CONSTRAINED TASK SETS

Both the approach in Section 5 based on window transferring and the approach in Section 6 based on dynamic pattern variation have their own advantages. Generally the approach in Section 5 could be more energy efficient than the approach in Section 6. But it also has the problem that it could affect the schedulability of the task set adversely. On the other hand, the approach in Section 6 based on dynamic pattern variation could maintain the schedulability of the original task set. Therefore, to maximize the overall energy reduction while respecting the schedulability of the task set, a more reasonable way is to combine these two approaches in an integrated way. Specifically, we can partition the original task set into two subsets and schedule them with the approaches in Section 5 and Section 6 correspondingly. Then the problem becomes how to determine the subsets of tasks to be partitioned with window-constraint and the original (m, k)-constraint in a hybrid way to maximize the overall energy reduction. In this section, we first adopt a "branch-and-bound" method, similar to that in [57], to divide the task set $\mathcal{T}$ into two parts, *i.e.*, the subset $\mathcal{A}$ in which the tasks will be partitioned based on window-constraints and the subset $\mathcal{B}$ in which the tasks will be partitioned based on the original (m, k)-constraints. Before presenting our approach in detail, we first introduce a sufficient condition for checking the feasibility of such kind of hybrid task sets consisting of subsets $\mathcal{A}$ and $\mathcal{B}$, which is summarized in the following theorem.

THEOREM 7.1. *Given periodic task sets $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_N\}$. Let $\mathcal{B}$ contains the subset of tasks with mandatory jobs determined with the original (m, k)-constraints and $\mathcal{A}$ be the subsets of tasks with mandatory jobs determined through the corresponding window constraint $m_i/\frac{(m_i+k_i)}{2}$ for task τ_i in it. Let L be the ending point of the first busy period for executing the mandatory jobs only and $LCM(P_i)$ be the* least common multiple *of P_i, $i = 1, 2, ..., N$. Then $\mathcal{T}$ is schedulable if for any mandatory job absolute deadline $d \leq \min\{L, LCM(P_i)\}$.*

$$d \geq \sum_{\tau_a \in \mathcal{A}} \lceil \frac{m_a}{\lfloor \frac{(m_a+k_a)}{2} \rfloor} \rceil \lceil \frac{d - D_a}{T_a} \rceil^+ \rceil C_a + \sum_{\tau_b \in \mathcal{B}} \lceil \frac{m_b}{k_b} \rceil \lceil \frac{d - D_b}{T_b} \rceil^+ \rceil C_b \tag{25}$$

The right side of equation (25) represents an upper bound of the total work demand from the mandatory jobs in $\mathcal{A}$ and in $\mathcal{B}$ with absolute deadlines less than or equal to d. The proof for this theorem could be done in a similar way to that for Theorem 1 in [14] and is thus omitted.

Based on Theorem 7.1, our branch-and-bound approach is presented in Algorithm 3.

Algorithm 3 Partitioning the task set $\mathcal{T}$ into subsets $\mathcal{A}$ and $\mathcal{B}$.

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

FUNCTION Try-Window-Constraint ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{Z}$, E_{bd})
for each task $\tau_i \in \mathcal{B}$ **do**
    Re-determine the mandatory jobs of τ_i based on the *window-constraint* that can be converted to its original (m, k)-constraint;
    Remove τ_i from $\mathcal{B}$ and put it into $\mathcal{A}$;
    if $\mathcal{A} \cup \mathcal{B}$ is schedulable **then**
        Compute the expected energy saving E_{save} according to Equation (26);
        if $E_{save} > E_{bd}$ **then**
            $E_{bd} = E_{save}$; $\mathcal{Z} = \mathcal{A} \cup \mathcal{B}$;
        end if
        Try-Window-Constraint ($\mathcal{A}$, $\mathcal{B}$, $\mathcal{Z}$, E_{bd});
    else
        Restore the job patterns of τ_i to be based on its original (m, k)-constraint and put it back to $\mathcal{B}$;
    end if
end for

From Algorithm 3, our approach determines task by task if the mandatory main/backup jobs of each task should be determined based on the original (m, k)-constraint or based on the window-constraint (to be converted). When Algorithm 3 is finished, it is possible to reach certain hybrid configuration in which the tasks in $\mathcal{A}$ are partitioned based on the window-constraints, while the tasks in $\mathcal{B}$ are still partitioned based on their original (m, k)-constraints. And the resulting configuration should be the one that could maximize the expecAed and $\mathcal{B}$ energy saving E_{save} for the tasks in $\mathcal{A}$.

Note that in Algorithm 3, with the N_i value determined based on Lemma 5.2, when estimating the expected energy saving for the tasks in $\mathcal{A}$ subject to the transient faults, we should incorporate the possible extra energy consumption of running the backup job(s) for the mandatory main job(s) overlapped with the backup job(s) of some other (already) failed mandatory job(s), if any, as well. Specifically,

$$\begin{aligned} E_{save} &= \sum_{\tau_i \in \mathcal{A}} \{2m_i C_i \frac{LCM_{\tau_i \in \mathcal{A}}(k_i T_i, \frac{(m_i+k_i)}{2} T_i)}{k_i}\} \\ &- \sum_{\tau_i \in \mathcal{A}} \{[m_i C_i + N_i \lambda_i(s_{max}) C_i] \frac{LCM_{\tau_i \in \mathcal{A}}(k_i T_i, \frac{(m_i+k_i)}{2} T_i)}{\frac{(m_i+k_i)}{2}}\} \end{aligned} \quad (26)$$

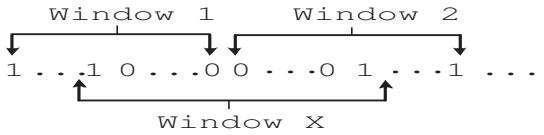


Fig. 11. The worst case of job patterns under two consecutive windows.

where $\lambda_i(s_{max}) = (1 - \gamma(J_{ij}))$, representing the average job fault rate for task τ_i at the maximal processor speed s_{max}.

Note that in the above Equation (26), the first part of it represent the worst case total energy consumption (without cancellation) within the hyper period before window transferring and the second part of it represent the worst case total energy consumption within the hyper period after window transferring.

7.1 Dealing with tasks with $(m_i + k_i)$ non-dividable by 2

It has not escaped from our attention that, for certain task τ_i, if $(m_i + k_i)$ for it is not dividable by 2, the value of $\frac{(m_i+k_i)}{2}$ for the corresponding window will not be an integer and cannot be used as the window length. In this case, we need to determine the window length in a more flexible way, *i.e.*, the window length should be determined as $\lfloor \frac{(m_i+k_i)}{2} \rfloor$ and $\lceil \frac{(m_i+k_i)}{2} \rceil$ alternatively. For example, for task τ_i with original (m, k)-constraint of $(3, 6)$, its corresponding window length under the window constraint should be $\lfloor \frac{(3+6)}{2} \rfloor = 4$ and $\lceil \frac{(3+6)}{2} \rceil = 5$ alternatively. As a result, its job patterns under window-constraint should be "1110" and "11010" alternatively, which will be "11101101011011010... ". It is easy to verify that this new job pattern can still satisfy the original (m, k)-constraint of $(3, 6)$. In the following, we will formulate that into a lemma.

LEMMA 7.2. *For any task τ_i with original (m, k)-constraint of (m_i, k_i), if its job patterns are determined under the separate windows using window-constraints of $m_i/\lfloor \frac{(m_i+k_i)}{2} \rfloor$ and $m_i/\lceil \frac{(m_i+k_i)}{2} \rceil$ alternatively, the original (m, k)-constraint of τ_i will be satisfied.*

PROOF. For brevity we only need to check the first two windows as the remaining windows will be a repeated version of the first two.

Obviously, when $(m_i + k_i)$ is dividable by 2, the result is trivially true from Lemma 5.1. So we only need to consider the case when $(m_i + k_i)$ is not dividable by 2. Under this case, with the job patterns determined above, the worst case will be the case when all optional jobs from two consecutive windows are clustered together, as shown in Figure 11. It is easy to see that the total number of optional jobs in Window 1 and Window 2 are $(\lfloor \frac{(m_i+k_i)}{2} \rfloor - m_i)$ and $(\lceil \frac{(m_i+k_i)}{2} \rceil - m_i)$, respectively. So in the worst case the total number of optional jobs across the border of two consecutive windows are

$$(\lfloor \frac{(m_i + k_i)}{2} \rfloor - m_i) + (\lceil \frac{(m_i + k_i)}{2} \rceil - m_i) = k_i - m_i \tag{27}$$

Note that since all optional jobs in two consecutive windows are already clustered together, the remaining jobs in the two consecutive windows are all mandatory jobs. As such, for any sliding window X of k_i jobs, obviously the worst case is that window X contains all the optional jobs across the border of two consecutive windows. As mentioned above, the other jobs in window X can only be mandatory jobs. So the number of mandatary jobs in window X is $k_i - (k_i - m_i) = m_i$, which satisfies the original (m_i, k_i)-constraint.

Note that in the above case we assume all optional jobs are clustered together, which is the worst case among all kinds of job patterns. For E-pattern, due to the even distribution of the mandatory/optional jobs, this case might not

happen, which means the number of optional jobs in any sliding window X could be less than $(k_i - m_i)$. Therefor the number of mandatory jobs in any sliding window of k_i jobs will be at least m_i. □

Algorithm 4 The online algorithm for (m, k)-constrained tasks

1: **For the primary processor:**
2: **if** MJQ is not empty **then**
3: Let J_i be the job with highest priority in MJQ.
4: **if** $\tau_i \in \mathcal{A}$ **then**
5: Execute J_i based on regular EDF scheme;
6: **else**
7: Execute J_i following the rationale in Algorithm 1;
8: **end if**
9: Let $J_i^{'}$ be the job of task $\tau_i^{'}$ in the other processor within the same time frame as J_i;
10: **if** J_i is completed successfully **then**
11: **if** $J_i^{'}$ is not the backup job of some other failed mandatory main job **then**
12: Cancel J_i's corresponding job in the other processor and add the residue time budget to the slack queue $\mathcal{S}$;
13: **end if**
14: **else if** $S_i(t) \geq C_i$ **then**
15: Reserve a recovery job R_i for job J_i and insert it into MJQ;
16: **else**
17: NTM' = the earliest arrival of upcoming mandatory job(s) in the other processor;
18: **if** $J_i^{'}$ is optional and $C_i \leq (\min\{NTM, d_i\} - t_{cur})$ **then**
19: Insert $J_i^{'}$ into the OJQ_A;
20: **end if**
21: **end if**
22: **else if** OJQ_A is not empty **then**
23: Select J_o in OJQ_A with the earliest deadline among jobs in OJQ_A
24: Run J_o non-preemptively;
25: **if** J_o is completed successfully **then**
26: Cancel the backup job of the failed mandatory job corresponding to J_o;
27: **end if**
28: **else if** OJQ_B is not empty **then**
29: NTM = the earliest arrival of upcoming mandatory job(s);
30: Select J_o in OJQ_B with the maximal window energy saving $\delta(WE_i)$ among jobs in OJQ_B with $C_o \leq (\min\{NTM, d_o\} - t_{cur})$;
31: **if** $J_o \neq \emptyset$ **then**
32: Run J_o non-preemptively;
33: **if** J_o is completed successfully **then**
34: Restart the E^R-pattern of task τ_o from the next job position following J_o;
35: **end if**
36: **end if**
37: **else**
38: t_{cur} = the current time;
39: **if** $(NTM - t_{cur}) > t_{sd}$ **then**
40: Shut down the processor and set up its wake-up timer to be $(NTM - t_{cur})$;
41: **end if**
42: **end if**
43: **For the spare processor:**
44: **if** $MJQ^{'}$ is not empty **then**
45: Repeat lines 3-21;
46: **else if** $OJQ_A^{'}$ is not empty **then**
47: Repeat lines 23-27;
48: **else**
49: Repeat lines 38-41;
50: **end if**

7.2 The overall online algorithm

Based on the output from Algorithm 3 and the above information, our overall online algorithm for general (m, k)-constrained tasks can be implemented in Algorithm 4.

As shown in Algorithm 4, in the primary processor, three job ready queues, which corresponds to three priority levels, are maintained: the mandatory job queue (MJQ), the optional job queue (OJQ_A) for the tasks in $\mathcal{A}$, and the optional job queue (OJQ_B) for the tasks in $\mathcal{B}$. The jobs in the MJQ are in the highest priority level and the jobs in the OJQ_B are in the lowest priority level. Upon release, a job of task τ_i is inserted into the MJQ if it is a mandatory job or a reserved recovery job, regardless whether τ_i belongs to $\mathcal{A}$ or $\mathcal{B}$. On the other hand, an optional job J_i of task τ_i is inserted into the OJQ_A if and only if $\tau_i \in \mathcal{A}$ and J_i's corresponding job in the other processor is a failed mandatory main job whose reclaimable slack time $S_i(t)$ is not long enough for reserving a recovery job for it. Meanwhile, J_i is inserted into the OJQ_B if and only if $\tau_i \in \mathcal{B}$ and J_i is executable based on the output from Algorithm 2 , *i.e.*, $Executable(J_{ij})$ is true. Otherwise J_i will be skipped. In this way we can avoid executing excessive number of optional jobs in $\mathcal{B}$. In the spare processor, things will be a little different as, to avoid executing too many optional jobs (which could consume more energy than necessary), the optional jobs of the tasks in $\mathcal{B}$ should not be executed there. As such, in the spare processor, only the mandatory backup job queue (represented as $MJQ^{'}$) and the optional job queue ($OJQ_A^{'}$) for the tasks in $\mathcal{A}$ will be maintained.

During runtime, for the tasks in $\mathcal{A}$, their mandatory jobs will be executed based on regular EDF scheme and the optional jobs will be invoked and executed only when necessary (lines 23-27). For the tasks in $\mathcal{B}$, some optional jobs will be selectively executed in the primary processor with dynamic pattern variation (lines 29-36). Note that if an optional job cannot be completed timely, it is not energy beneficial and therefore should not be invoked at all. As such, an optional job is regarded as eligible only if it could surely be finished before the earliest arrival time of the upcoming mandatory jobs (line 30). If there are multiple eligible optional jobs from OJQ_B, the window energy saving $\delta(WE_i)$ for each eligible optional job will be calculated, where $\delta(WE_i)$ is defined as $(WE_i' - WE_i)$. The one with the maximal window energy saving $\delta(WE_i)$ should be selected for execution. Moreover, once a selected optional job is invoked, it should be executed non-preemptively to ensure that it could be finished timely.

Once an optional job in $OJQ_A^{'}$ is completed successfully, the backup job of the failed mandatory main job corresponding to the optional job will be canceled (lines 25-27). Once an optional job in $OJQ_B^{'}$ is completed successfully, the patterns of the future jobs should be varied by restarting the E^R-pattern of the same task in both the primary and the spare processors from the next job position. Note that if no optional jobs are available, the mandatory jobs in $\mathcal{B}$ can still be executed following the rationale in Algorithm 1 based on the adaptive delay policies in Section 8.1.1 (lines 7 and 46). The only issue is, in this case, since EDL scheme is not applicable for task sets with dynamic pattern variation, the delayed release time of any mandatory job J_i of task τ_i should be calculated based on the following sufficient conditions:

LEMMA 7.3. *[58] Given task set $\mathcal{T}$ with all of its mandatory jobs schedulable, if the release time of any mandatory job J_i is delayed to $(r_i + Y_i)$, no mandatory job will miss its deadline, where*

$$Y_i = D_i - R_i \tag{28}$$

and R_i is the worst case response time of task τ_i's mandatory jobs which could be computed offline with the approach in [59].

LEMMA 7.4. *Assuming at time t, let J_i be the current mandatory job and $\mathcal{J}'$ be the set of the unfinished mandatory jobs and/or the upcoming mandatory jobs with arrive times later than t. Also let T_{bound} be the delay bound (*i.e.*, the earliest deadline for the mandatory jobs in $\mathcal{J}'$) for $\mathcal{J}'$. Then no mandatory job in $\mathcal{J}'$ will miss its deadline if the execution of job J_i is delayed to $t_d(J_i)$, where*

$$t_d(J_i) = \min_{J_q \in \mathcal{J}_s} (d_q^* - \sum_{J_j \in hp(J_q)} C_j), \tag{29}$$

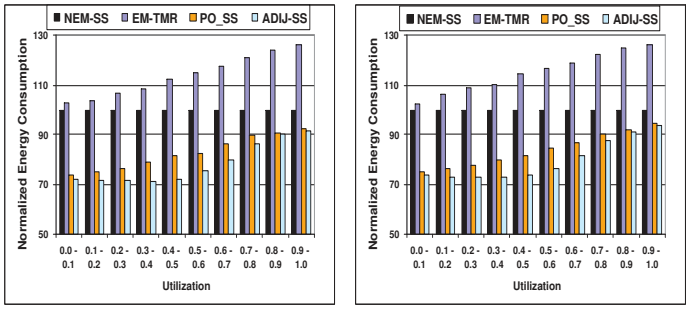


Fig. 12. The results in the presence of (a) No faults; (b) System faults.

where $\mathcal{J}_s$ consists of the unfinished mandatory jobs from $\mathcal{J}'$ with arrival times earlier than T_{bound} but later than t, $hp(J_j)$ are the jobs with priorities higher than or equal to that of J_j, and

$$d_q^* = \min_k(d_q, r_k + Y_k), \forall J_k \in \mathcal{J}', J_k \notin \mathcal{J}_s \text{ and } d_k > d_q. \tag{30}$$

where Y_k is computed through Equation (28).

The main difference between Lemma 7.4 and Theorem 1 in [50] is that our sufficient condition in Lemma 7.4 can be used to compute the delayed release time for the current mandatory job not only when the processor is idle, but also when the processor is busy at time t. Moreover, the effective deadline for a mandatory job is relaxed from the earliest arrival time of the next lower priority mandatory job further by a time interval of Y_k which will allow the current mandatory job to be delayed further. The proof of Lemma 7.4 can be done in a similar way to that of Theorem 1 in [50] and is thus omitted.

Moreover, it is easy to see that at any time t, the execution of any mandatory job can be delayed by $S_i(t)$ units, where $S_i(t)$ is the slack time from $\mathcal{S}$ with priorities higher than or equal to J_i.

Based on the analysis in the above sufficient conditions, the delayed release time of a mandatory job J_i in Algorithm 4 could be re-calculated as $\hat{r}_i = \max\{(r_i + Y_i),\ t_d(J_i),\ S_i(t)\}$ safely.

The online complexity of Algorithm 4 mainly comes from scheduling the mandatory and optional jobs. Since at anytime there are at most n jobs in the MJQ or in the OJQ_A (OJQ_B), its complexity is $O(k_iN)$, which is linear since k_i is usually a small constant integer.

Moreover, to ensure that the (m, k)-constraint of the tasks be satisfied, we have the following theorem (proof omitted):

THEOREM 7.5. *Let task set $\mathcal{T}$ be scheduled with Algorithm 4. The (m, k)-deadlines for $\mathcal{T}$ can be ensured if $\mathcal{T}$ is schedulable under E-pattern.*

8 EVALUATION

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

The processor model used in our simulations is based on the Free-scale PowerQUICC III integrated Communications Processor MPC8536E [60], similar to the one used in [61]. According to the data sheet in [60], the typical power consumption of MPC8536E running under the maximal frequency is 4.7 Watt (with a core frequency of

1500 MHz and core voltage of 1.1 V). The idle power P_{idle} is about 0.6 Watt. Since the transition overheads are not mentioned in the data sheet, we assumed the shut-down/wake-up time overhead $t_o = 1$ millisesond and energy overhead $E_o = 0.6$ mJule. Therefore the minimal shut-down interval t_{sd} will be calculated as 1 millisesond.

8.1 Evaluation based on synthesized task sets

8.1.1 Simulation results for task sets with $(1, 1)$*-constraint.* Four different approaches were studied. The first approach (NEM_{SS}) executed the jobs in the primary and the spare processors concurrently without energy management. We used its results as the reference. The second approach (*EM-TMR*) scheduled the task with the Triple Modular Redundancy scheme from [46] but without applying DVFS. The third approach (PO_{SS}) scheduled the tasks with the preference oriented scheme from [10] with task preference randomly determined and without applying DVFS. The fourth approach ($ADIJ_{SS}$) is our approach proposed in Section 4 based on adaptive delay policies on individual jobs in both the primary and the spare processors.

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

In the first set, we check the energy performance when no fault occurred within the hyperperiod. The result is shown in Figure 12(a).

From Figure 12(a), it is not surprising that the energy consumption of *EM-TMR* is always the highest among all approaches mainly because it adopted the Triple Modular Redundancy scheme to provide fault tolerance, *i.e.*, to employ and analyze the results of three separate executions of the same job to determine the correct one. Moreover, one can immediately see that, by adopting the adaptive delay policies on each main/backup job individually, $ADIJ_{SS}$ can achieve much better energy efficiency than the other two previous approaches, *i.e.*, NEM_{SS} and PO_{SS}, in most utilization intervals. The energy reduction by $ADIJ_{SS}$ over PO_{SS} can be up to 11%. The main reason is that, in this scenario, by executing the main/backup jobs in a more flexible way, $ADIJ_{SS}$ can help reduce the overlapped execution between the main jobs and their backup jobs more efficiently, therefore saving more energy.

In the second set, we assumed the system could be subject to permanent/transient faults. The permanent fault could occur randomly at most once during the hyperperiod. The transient fault model is similar to that in [4] by assuming Poisson distribution with an average fault rate of 10^{-6} per millisecond. The result is shown in Figure 12(b).

As seen, in this scenario, the energy consumption of *EM-TMR* is still much higher than the other approaches for the same reason as stated above. Meanwhile the energy reduction by our new approaches, *i.e.*, $ADIJ_{SS}$, over PO_{SS} could be affected slightly by the possible permanent/transient faults, but can still achieve up to 9% energy saving, thanks to the adaptive executions of the jobs under individual/flexible delay.

8.1.2 Simulation results for task sets with general (m, k)*-constraint.* In this part, we studied five approaches. The first approach (NEM_{MKE}) statically determined the job patterns based on *E*-pattern. And the mandatory jobs in the primary and the spare processors were executed concurrently without delay. We used its results as the reference. The second approach (*EM-TMR*) also determined the job patterns based on *E*-pattern and the mandatory jobs were scheduled with the Triple Modular Redundancy scheme from [46] but no DVFS was applied. The third approach ($MKSS_{Selective}$) also determined the job patterns based on *E*-pattern first but selectively executed the optional jobs using the approach in [48]. The fourth approach ($WCMK_{SS}$) is our proposed approach presented in the conference version [62]. The fifth approach ($WCMK_{EN}$) is our new combined approach with the enhanced techniques proposed in this journal version.

The periodic task sets are generated in the same way as in Section 8.1.1 but with m_i and k_i values for the (m, k)-constraint randomly generated between 2 and 10 ($k_i > m_i$). Since when the total (m, k)-utilization, *i.e.*,

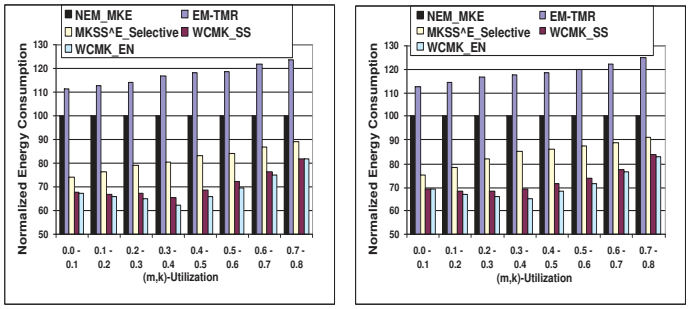


Fig. 13. The results in the presence of (a) No faults; (b) System faults.

$\sum_i \frac{m_i C_i}{k_i P_i}$ is larger than 0.8, it is hard for the task sets to be schedulable, we mainly checked the task sets with (m,k)-utilization between 0.0 to 0.8.

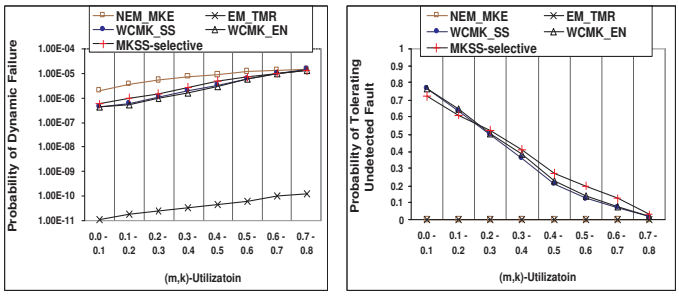
Firstly, we inspect the energy consumption of the different approaches. We performed two sets of experiments. In the first set, we assumed no fault occurred within the hyperperiod. The result is shown in Figure 13(a).

From Figure 13(a), it is not surprising that under the general (m,k)-constraint *EM-TMR* still incurred much higher energy consumption than all the other approaches based on standby-sparing because it adopted the triple modular redundancy scheme in which one additional copy of the job was executed (to generate the dominant result), incurring more energy consumption. Meanwhile, it is easy to see that $WCMK_{SS}$ and $WCMK_{EN}$ can achieve much better energy performance than all the other approaches. The energy reduction by $WCMK_{SS}$ over $MKSS_{Selective}$ can be up to 20%. The main reason is that, in this scenario, by partitioning the jobs based on window-constraints (that could be converted to the original (m,k)-constraints) first, $WCMK_{SS}$ can help minimize the overlapped execution between the mandatory and backup jobs in two processors more efficiently. Moreover, with the enhanced techniques proposed in this journal version, $WCMK_{EN}$ can boost the energy saving further by about 3% thanks to the more adaptive techniques proposed in this journal version.

In the second set, we assumed the system was subject to permanent and/or transient faults with the same fault rate as in Section 8.1.1. The result is shown in Figure 13(b).

As seen, under this scenario, similar to the above results in Figure 13(a), the energy consumption of *EM-TMR* is still the highest among all approaches compared. Meanwhile, the energy saving achievable by our new approach, *i.e.*, $WCMK_{SS}$ over $MKSS_{Selective}$ is even better, *i.e.*, up to 22%. Similar to the fault-free case, our enhanced approach proposed in this journal version, *i.e.*, $WCMK_{EN}$ can boost the energy saving by additionally 3%. This is mainly because the energy efficiency of $MKSS_{Selective}$ is highly dependant on the successfully execution of optional jobs and the dynamic pattern variation based on it. As such, when the system fault(s) occurred, the dynamic pattern variation procedure in $MKSS_{Selective}$ could be affected significantly. Different from that, in our new approaches, *i.e.*, $WCMK_{SS}$ and $WCMK_{EN}$, the execution of the optional jobs under dynamic pattern shifting only partially contributed to the overall energy reduction. A more significant part of the energy saving comes from our more flexible job partitioning strategy based on window-constraint (that could be transferred to the original (m,k)-constraint) as well as the adaptive executions of the mandatory main/backup jobs based on flexible delay when necessary.

Next, we inspected the reliability of the system by the different approaches. Here in order to reflect the system reliability in a more straightforward way, we checked the *probability of dynamic failure* (denoted as PoDF) of the different approaches. The PoDF is defined as the probability for the system to have any task in it with a window



(a) (b)

Fig. 14. The comparisons on (a) Probability of dynamic failure; (b) Capability of tolerating one undetected fault.

of k_i consecutive jobs having less than m_i jobs completed successfully. We assumed the system was subject to permanent and/or transient faults with same fault rate as in Section 8.1.1. The result is shown in Figure 14(a).

As can be seen from Figure 14(a), in all utilization intervals, as expected, the PoDF of *EM-TMR* is much lower than all the other approaches adopting standby-sparing. However, its energy cost is also much higher, as shown in Figure 13. Meanwhile, among all the approaches based on standby-sparing, the PoDFs of $WCMK_{SS}$ and $WCMK_{EN}$ are always the lowest. The effect is especially obvious when the (m,k)-utilization is modest. This is because under this scenario most tasks in $WCMK_{SS}$ and $WCMK_{EN}$ had chance to be partitioned into subset $\mathcal{A}$ via branch-and-bound in Algorithm 3. As mentioned in Section 5.2, after window transferring, the tasks in subset $\mathcal{A}$ contains a number of redundant mandatory jobs, which is very helpful in reducing the PoDF. As seen in Figure 14(a), compared with NEM_{MKE}, the reduction in PoDF is roughly one order of magnitude when the (m,k)-utilization is not very high. Meanwhile, it is interesting to see that the PoDF of $MKSS_{Selective}$ is also relatively lower than NEM_{MKE}. This is mainly because it selectively executed a number of optional jobs, which is practically helpful in reducing PoDF as well. However, the energy consumption of $MKSS_{Selective}$ is also much higher than $WCMK_{SS}$ and $WCMK_{EN}$ as shown in Figure 13. Moreover, since the reduction of PoDF in $WCMK_{SS}$ and $WCMK_{EN}$ mainly comes from the pre-reserved redundant mandatory jobs of the tasks in subset $\mathcal{A}$, it is predictable based on Lemma 5.3, whereas the reduction of PoDF in $MKSS_{Selective}$ mainly comes from the dynamically executed optional jobs, which is not predictable.

In the following, by considering the possibility of undetected faults, we also inspect the probability of tolerating at least one undetected fault in any window of k_i consecutive jobs by the different approaches. The result is shown in Figure 14(b).

As can be seen from Figure 14(b), it is not surprising that the probabilities for NEM_{MKE} and *EM-TMR* to tolerate undetected faults are zero because they adopted the original static E-patterns which defined a minimal set of mandatory jobs that "just" satisfied the given (m,k)-constraints (therefore any undetected fault could cause a dynamic failure). In contrast to them, $WCMK_{SS}$ and $WCMK_{EN}$ have much higher capability of tolerating at least one undetected fault in any window inspected, especially when the (m,k)-utilization is modest, for the same reason as stated above. Meanwhile, it is noted that the practical probability for $MKSS_{Selective}$ to tolerate at least one undetected fault in any window is also quite good (in some utilization intervals even slightly higher than those by $WCMK_{SS}$ and $WCMK_{EN}$). However, similar to the above argument, its energy consumption is also much higher. Moreover, since it is based on the execution of optional jobs, its performance under this scenario is not predictable whereas the performances of $WCMK_{SS}$ and $WCMK_{EN}$ are predictable because the redundancy of the mandatory jobs of the tasks in subset $\mathcal{A}$ could be estimated based on Lemma 5.3.

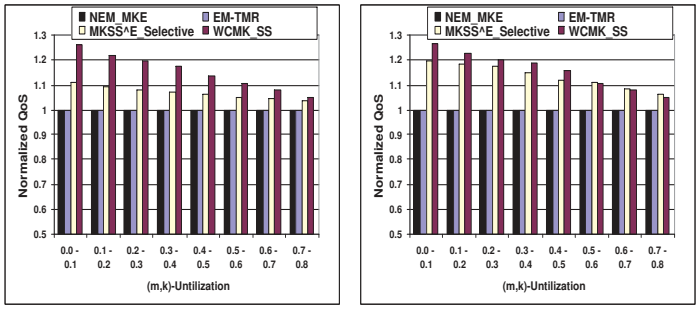


Fig. 15. Comparisons on the QoS for systems in the presence of (a) No faults; (b) System faults.

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

In the first set, we checked the QoS when no fault occurred within the hyperperiod. The results were normalized to that by *SSNEM* and shown in Figure 15(a). From Figure 15(a), we can see that our newly proposed approach, *i.e.*, $WCMK_{SS}$ could achieve much better QoS levels than the existing approaches. Compared with *EM-TMR* and $MKSS_{Selective}$, the maximal QoS improvement could be up to 28% and 16%, respectively. This is mainly due to the redundant mandatory jobs executed under window transferring in $WCMK_{SS}$.

In the second set, we assumed the system could be subject to permanent/transient faults with the same fault rate as in the previous groups of tests. The result is shown in Figure 15(b).

From Figure 15(b), the QoS improvement subject to faults by our newly proposed approach, *i.e.*, $WCMK_{SS}$ over the *EM-TMR* is quite similar to that when no fault ever occurred, for the same reason as stated above. Meanwhile, it is also interesting to see that under this scenario the QoS of $MKSS_{Selective}$ could be quite close to that of $WCMK_{SS}$ in certain utilization intervals. However, as shown in Figure 13(b), its energy consumption in this case is also much higher due to excessive number of jobs executed.

8.2 Evaluation based on real world benchmark

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

8.2.1 Simulation results for task sets with $(1, 1)$*-constraint.* In this part we performed tests on two real world applications, *i.e.*, INS (Inertial Navigation System) [63] and CNC (Computerized Numerical Control) machine controller [64]. The timing parameters such as the deadlines, periods, and execution times were adopted from the practical applications directly [63, 64]. For the fault model we adopted the same model as used for the synthesized task sets.

We performed two sets of experiments to inspect the energy consumption of the different approaches.

In the first set, we assumed no fault occurred within the hyperperiod. The result is shown in Figure 16(a).

From Figure 16(a), it is obvious that for both applications the energy consumption of *EM-TMR* is still the highest among all approaches, mainly due to the Triple Modular Redundancy scheme adopted in it. On the other hand, by adopting the adaptive delay policies on each main/backup job individually, $ADIJ_{SS}$ can achieve much better energy efficiency than the other two previous approaches, *i.e.*, NEM_{SS} and PO_{SS} for both applications. The energy reduction by $ADIJ_{SS}$ over PO_{SS} can be around 9% for INS and 5% for CNC, respectively, thanks to the adaptive executions of the jobs under individual/flexible delay.

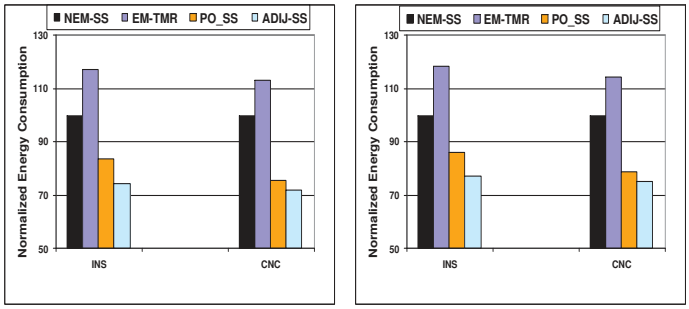


Fig. 16. The results based on real world benchmarks INS [63] and CNC [64] in the presence of (a) No faults; (b) System faults.

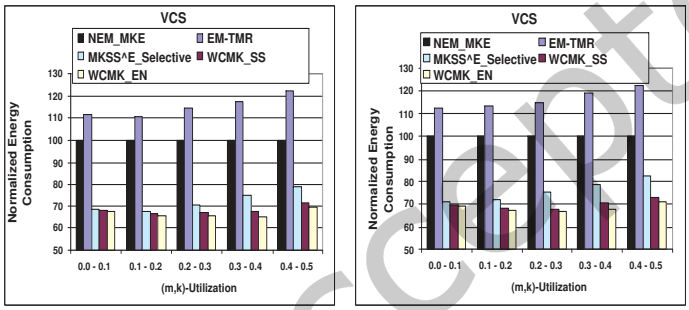


Fig. 17. The results based on real world benchmark VCS [13] in the presence of (a) No faults; (b) System faults.

In the second set, we assumed the system could be subject to permanent and/or transient faults with the same fault rate as in Section 8.1.1 in the main manuscript. The result is shown in Figure 16(b).

As seen, in this scenario, for both applications the energy consumption of $EM\text{-}TMR$ is still much higher than the other approaches for the same reason as stated above. Meanwhile, the energy consumption of our newly proposed approach, *i.e.*, $ADIJ_{SS}$, is still much lower than the other two previous approaches, namely NEM_{SS} and PO_{SS}, with energy reduction of around 10% for INS and 4% for CNC, for the same reason as stated above.

8.2.2 Simulation results for task sets with general (m,k)*-constraint.* The test is based on another real world benchmark: VCS (Vehicle Control System) [13]. The timing parameters such as the deadlines, periods, and execution times were adopted from the practical application directly [13]. The m_i and k_i values for the (m,k)-constraint were randomly generated between 2 and 10 ($k_i > m_i$). We also conducted two sets of tests.

In the first set, we checked the energy performance when no fault occurred within the hyperperiod. The result is shown in Figure 17(a).

From Figure 17(a), it is easy to see that for VCS application, the energy consumption of $EM\text{-}TMR$ is much higher than all the other approaches based on standby-sparing for the same reason as stated above. Meanwhile, $WCMK_{SS}$ and $WCMK_{EN}$ can achieve much better energy performance than all the other approaches. The energy reduction by $WCMK_{SS}$ and $WCMK_{EN}$ over $MKSS_{Selective}$ can be up to 9% and 12%, respectively, which conforms to our analysis on synthesized task sets as well.

In the second set, we assumed the system could be subject to permanent and/or transient faults with same fault rate as in Section 8.1.1. The result is shown in Figure 17(b).

As seen, under this scenario, the energy savings achievable by our newly proposed approaches, *i.e.*, $WCMK_{SS}$ and $WCMK_{EN}$ over the previous approach are even better. The energy reduction by $WCMK_{SS}$ and $WCMK_{EN}$ over $MKSS_{Selective}$ can be up to 11% and 13%, respectively. This is mainly due to the fact that the energy efficiency of $MKSS_{Selective}$ is highly dependant on the successfully execution of optional jobs and the dynamic pattern variation based on it. As such, when the system fault(s) occurred (to the optional jobs), the dynamic pattern variation procedure in $MKSS_{Selective}$ could be affected significantly. Different from that, in our new approaches, *i.e.*, $WCMK_{SS}$ and $WCMK_{EN}$, the execution of the optional jobs under dynamic pattern shifting only partially contributed to the overall energy reduction. A more significant part of the energy saving in them came from our more flexible mandatory job shifting strategy based on window-constraint (that could be transferred to the original (m,k)-constraint) as well as the adaptive executions of the mandatory main/backup jobs based on flexible delay when necessary.

Overall, the evaluation results based on both synthesized systems and real world applications have clearly demonstrated the effectiveness of our approaches in saving energy while satisfying the (m,k)-constraints and assuring fault tolerance through standby-sparing.

9 CONCLUSION

Fault-tolerance, energy consumption, and quality of service are becoming increasingly critical factors in the design of pervasive computing systems. In this paper, we focuses on exploring methods that can simultaneously address the above three important issues under the standby-sparing mechanism with the purpose of providing fault tolerance subject to both permanent and transient faults. Due to its duplicate executions of the real-time jobs/tasks, the energy consumption of a standby-sparing system could be quite high. To save energy under standby-sparing, we proposed three novel scheduling schemes: the first one is for $(1,1)$-constrained tasks, and the second one and the third one are for general (m,k)-constrained tasks which require that among any k consecutive jobs of a task no more than $(k-m)$ out of them could miss their deadlines. Based on the second and the third approaches, a combined approach is also proposed to maximize the overall energy reduction while respecting the schedulability of the task sets. Through extensive evaluations and performance analysis, our results demonstrate that the proposed techniques significantly outperform the existing ones in energy reduction while assuring (m,k)-constraints and fault tolerance under standby-sparing.

ACKNOWLEDGMENTS

This work is partly supported by the U.S. NSF under grants ECCS 2302651, HRD 2135345, CNS/SaTC 2039583, HRD 1828811, CMMI 2240407, the Swedish Research Council Grant No. 2023-04485, and by the Research Institute For Tactical Autonomy, A University Affiliated Research Center (RITA-UARC) of the U.S. Department of Defense at Howard University under Contract Number FA955023D0001 with the U.S. Air Force Office of Scientific Research.

REFERENCES

[1] M. Johnson, D. Somasekhar, L.Y. Choiu, and K. Roy. 2002. Leakage Control With Efficient Use of Transistor Stacks in Single Threshold CMOS. *IEEE Trans. on VLSI* 10, 1 (February 2002), 1–5.

[2] Le Yan, Jiong Luo, and Niraj K. Jha. 2003. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. *ICCAD* (2003).

[3] Xiliang Zhong and Cheng-Zhong Xu. 2008. System-wide energy minimization for real-time tasks: Lower bound and approximation. *ACM Trans. Embed. Comput. Syst.* 7, Article 28 (May 2008), 24 pages. Issue 3.

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

[5] 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).
[6] 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.
[7] 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.
[8] 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.
[9] M. A. Haque, H. Aydin, and D. Zhu. 2011. Energy-aware Standby-Sparing Technique for periodic real-time applications. In *ICCD*.
[10] 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).
[11] Houssine Chetto and Maryline Chetto. 1989. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transction On Software Engineering* 15 (1989).
[12] D. Shin, J. Kim, and S. Lee. 2001. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design and Test of Computers* 18, 2 (March-April 2001).
[13] 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. DOI: http://dx.doi.org/10.1109/TII.2006.875511
[14] 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.
[15] G. Bernat and A. Burns. 2001. Weakly hard real-time systems. *IEEE Trans. on Comp.* 50, 4 (April 2001), 308–321.
[16] 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.
[17] 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.
[18] 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.
[19] G. Bernat and A. Burns. 1997. Combining (n,m)-hard deadlines and dual priority scheduling. In *RTSS*.
[20] Maryline Chetto. 2015. Graceful Overload Management in Firm Real-Time Systems. *Journal of Information Technology and Software Engineering* 5, 3 (2015), 1–3.
[21] 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 (RTNS '15)*. 237–246.
[22] 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.
[23] 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. DOI: http://dx.doi.org/10.1145/3126497
[24] T. A. AlEnawy and H. Aydin. 2005. Energy-Constrained Scheduling for Weakly-Hard Real-Time Systems. *RTSS* (2005).
[25] H. Kooti, N. Dang, D. Mishra, and E. Bozorgzadeh. 2012. Energy Budget Management for Energy Harvesting Embedded Systems. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. 320–329. DOI: http://dx.doi.org/10.1109/RTCSA.2012.38
[26] 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.
[27] 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.
[28] Dakai Zhu and H. Aydin. 2009. Reliability-Aware Energy Management for Periodic Real-Time Tasks. *Computers, IEEE Transactions on* 58, 10 (2009), 1382–1397.
[29] Mohsen Ansari, Sepideh Safari, Farimah Poursafaei, Mohammad Salehi, and Alireza Ejlali. 2018. AdDQ: Low-Energy Hardware Replication for Real-Time Systems through Adaptive Dual Queue Scheduling. *The CSI Journal on Computer Science and Engineering (JCSE)* 15 (04 2018), 31–38.
[30] Abhishek Roy, Hakan Aydin, and Dakai Zhu. 2017. Energy-efficient primary/backup scheduling techniques for heterogeneous multicore systems. In *2017 Eighth International Green and Sustainable Computing Conference (IGSC)*. 1–8. DOI: http://dx.doi.org/10.1109/IGCC.2017.8323569
[31] 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).

[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] 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.
[34] Mohsen Ansari, Mohammad Salehi, Sepideh Safari, Alireza Ejlali, and Muhammad Shafique. 2020. Peak-Power-Aware Primary-Backup Technique for Efficient Fault-Tolerance in Multicore Embedded Systems. *IEEE Access* 8 (2020), 142843–142857. DOI: http://dx.doi.org/10.1109/ACCESS.2020.3013721
[35] Mohammad A. Haque, Hakan Aydin, and Dakai Zhu. 2013. Energy management of standby-sparing systems for fixed-priority real-time workloads. In *2013 International Green Computing Conference Proceedings*. 1–10. DOI: http://dx.doi.org/10.1109/IGCC.2013.6604487
[36] 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.
[37] 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.
[38] 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.
[39] 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.
[40] 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.
[41] 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.
[42] 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.
[43] Mohsen Ansari, Sepideh Safari, Nezam Rohbani, Alireza Ejlali, and Bashir M. Al-Hashimi. 2023. Power-Efficient and Aging-Aware Primary/Backup Technique for Heterogeneous Embedded Systems. *IEEE Transactions on Sustainable Computing* (2023), 1–12. DOI: http://dx.doi.org/10.1109/TSUSC.2023.3282164
[44] Mohsen Ansari, Sepideh Safari, Sina Yari-Karin, Pourya Gohari-Nazari, Heba Khdr, Muhammad Shafique, Jörg Henkel, and Alireza Ejlali. 2022. Thermal-Aware Standby-Sparing Technique on Heterogeneous Real-Time Embedded Systems. *IEEE Transactions on Emerging Topics in Computing* 10, 4 (2022), 1883–1897. DOI: http://dx.doi.org/10.1109/TETC.2021.3120084
[45] Sina Yari-Karin, Roozbeh Siyadatzadeh, Mohsen Ansari, and Alireza Ejlali. 2023. Passive Primary/Backup-Based Scheduling for Simultaneous Power and Reliability Management on Heterogeneous Embedded Systems. *IEEE Transactions on Sustainable Computing* 8, 1 (2023), 82–93. DOI: http://dx.doi.org/10.1109/TSUSC.2022.3186656
[46] 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.
[47] D. Zhu, R. Melhem, D. Mosse, and E. Elnozahy. 2004. Analysis of an energy efficient optimistic TMR scheme. In *Proceedings. Tenth International Conference on Parallel and Distributed Systems, 2004. ICPADS 2004.* 559–568.
[48] Linwei Niu and Dakai Zhu. 2020. Reliable and Energy-Aware Fixed-Priority (m,k)-Deadlines Enforcement with Standby-Sparing. *DATE* (2020).
[49] G. Koren and D. Shasha. 1995. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *RTSS*.
[50] Linwei Niu and G. Quan. 2004. Reducing both dynamic and leakage energy consumption for hard real-time systems. *CASES'04* (Sep 2004).
[51] Nam Kim, Todd Austin, D. Baauw, Trevor Mudge, Krisztián Flautner, Jie Hu, Mary Irwin, Mahmut Kandemir, and Vijaykrishnan Narayanan. 2004. Leakage Current: Moore's Law Meets Static Power. *Computer* 36 (01 2004), 68 – 75. DOI: http://dx.doi.org/10.1109/MC.2003.1250885
[52] D. K. Pradhan (Ed.). 1986. *Fault-tolerant Computing: Theory and Techniques; Vol. 2*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
[53] 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.
[54] G. Quan and X.(Sharon) Hu. 2000. Enhanced Fixed-Priority Scheduling with (m,k)-Firm Guarantee. In *RTSS*. 79–88.
[55] Mohammad A. Haque, Hakan Aydin, and Dakai Zhu. 2017. On Reliability Management of Energy-Aware Real-Time Systems Through Task Replication. *IEEE Transactions on Parallel and Distributed Systems* 28, 3 (2017), 813–825. DOI: http://dx.doi.org/10.1109/TPDS.2016.2600595
[56] Israel Koren and C. Krishna. 2020. *Fault-Tolerant Systems, 2nd Edition*. Morgan Kaufmann.
[57] Linwei Niu and Jia Xu. 2015. Improving Schedulability and Energy Efficiency for Window-constrained Real-time Systems with Reliability Requirement. *Journal of Systems Architecture* 61, 5 (May 2015), 210–226.

[58] Linwei Niu and Dakai Zhu. 2017. Reliability-aware scheduling for reducing system-wide energy consumption for weakly hard real-time systems. *Journal of Systems Architecture* 78 (2017), 30 – 54.
[59] J.A. Stankovic, M. Spuri, K. Ramamritham, and G. Buttazzo. 1998. *Deadline Scheduling for Real-Time Systems EDF and Related Algorithms*. Springer, Berlin, Germany.
[60] Freescale Semiconductor. 2015. MPC8536E PowerQUICC III Integrated Processor Hardware Specifications, Document Number: MPC8536EEC Rev. 7, 07/2015. *https://www.nxp.com/docs/en/data-sheet/MPC8536EEC.pdf* (2015).
[61] Muhammad Ali Awan and Stefan M. Petters. 2014. Race-to-halt energy saving strategies. *Journal of Systems Architecture* 60, 10 (2014), 796 – 815.
[62] Linwei Niu. 2021. Fault-Tolerant Energy Management for Real-Time Systems with Weakly Hard QoS Assurance. *IEEE International Conference on Computer Communications* (2021).
[63] A.Burns, K. Tindell, and A. Wellings. 1995. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering* 21 (May 1995), 920–934.
[64] N.Kim, M. Ryu, S. Hong, M. Saksena, C. Choi, and H. Shin. 1996. Visual assessment of a real-time system design: a case study on a CNC controller. In *RTSS*.