Fixed-Priority Scheduling for Reliable and Energy-Aware (m, k)-Deadlines Enforcement With Standby-Sparing

Linwei Niu, *Member, IEEE*, and Dakai Zhu, *Senior Member, IEEE*

***Abstract*—For real-time computing systems, energy efficiency, quality of service (QoS), and fault tolerance are among the major design concerns. In this work, we study the problem of reliable and energy-aware (m, k)-deadlines enforcement using standby sparing under the fixed-priority assignment. The standby-sparing systems adopt a primary processor and a spare processor to provide fault tolerance for both permanent and transient faults. In order to reduce energy consumption for such kinds of systems, we proposed two novel scheduling schemes under the QoS constraint of (m, k)-deadlines: one for task sets partitioned with deeply red pattern and one for task sets partitioned with evenly distributed pattern. The evaluation results demonstrate that our proposed approaches significantly outperformed the previous research in energy conservation while assuring (m, k)-deadlines and fault tolerance for real-time systems.**

***Index Terms*—Energy conservation, fault tolerance, fixed-priority (FP) scheduling, (m, k)-deadlines, standby sparing.**

I. Introduction

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

In recent years, extensive research studies (e.g., [1], [3]–[6]) have been reported in conserving energy for fault-tolerant real-time systems. Many of them have focused on dealing with transient faults. A widely adopted strategy is to use software redundancy, i.e., to reserve recovery jobs, whenever possible, for the jobs subject to transient faults. For mission-critical applications, such as nuclear plant control systems and heart pacemakers [7], permanent faults need to be dealt with by all means to avoid system failure. Otherwise, catastrophical consequences could occur. To address this issue, solutions adopting hardware redundancy are required. Among them, the *standby-sparing* technique has recently gained much attention in the research community [8]–[12]. 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: 1) a primary one and 2) 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, the execution of the main jobs in the primary processor and their corresponding backup jobs in the spare processor might need to be overlapped with each other. Thus, the total energy consumption could be quite considerable. Regarding that, some recent works have been reported to reduce energy (e.g., [8]–[10], [12]). 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 in the spare processor. With that in mind, in [10] and [11], approaches based on the dual-priority scheme were proposed for standby-sparing fixed-priority (FP) real-time systems. Their works are mainly focused on *hard* real-time systems.

In some real-time applications, occasional deadline missings are acceptable so long as the user perceived quality of service (QoS) can be assured at certain levels. For example, some remote monitoring applications that sense k times per minute while missing $(k - m)$ precisely distributed sensing samples might be acceptable but with some degraded QoS levels [13]. For such kinds of systems, the existing techniques solely based on hard real-time constraints are insufficient in dealing with energy conservation under fault tolerance and more advanced techniques incorporating the QoS model are desired.

A widely known deterministic QoS model is the (m, k) model [14]. To ensure the (m, k)-deadlines, Ramanathan [15] proposed to partition the jobs into *mandatory* ones and

Manuscript received September 5, 2020; revised December 5, 2020; accepted January 16, 2021. Date of publication February 23, 2021; date of current version February 21, 2022. This work was supported in part by NSF under Project HRD-1800403. This article was recommended by Associate Editor Z. Shao. *(Corresponding author: Linwei Niu.)*

Linwei Niu is with the Department of Electrical Engineering and Computer Science, Howard University, Washington, DC 20059 USA (e-mail: linwei.niu@howard.edu).

Dakai Zhu is with the Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249 USA.

Digital Object Identifier 10.1109/TCAD.2021.3061522

optional ones. The mandatory ones must be completed successfully, whereas the other ones could be optionally executed when necessary.

In this article, we study the problem of reliable and energy-aware (m, k)-deadlines enforcement with standby-sparing under FP assignment. To the best of our knowledge, this is the first work to combine (m, k)-deadlines and standby sparing to achieve better energy efficiency for real-time applications.

The remainder of the article is organized as follows. In Section II we discuss the related work. Section III presents the preliminaries. Section IV presents the motivations. Section V presents our proposed approach based on deeply red patterns. Section VII-B presents our approach based on evenly distributed patterns. In Section VII, we present our evaluation results. In Section VIII, we offer our conclusion.

II. Related Work

In the last few decades, a plenty of work has been done in integrating QoS assurance into scheduling for real-time systems. For systems with transient overloaded conditions, Chetto [16] explored scheduling algorithms for firm real-time systems. For mixed-criticality systems, Gettings *et al.* [17] and Brüggen *et al.* [18] proposed new approaches that can provide QoS-guarantee for low-criticality tasks. Moreover, for general FP weakly hard real-time systems, schedulability analysis based on the mixed integer linear programming (MILP) formulation is provided in [19]. Considering the given energy budget constraint, Alenawy and Aydin [20] 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.* [21] 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 have been presented in combining fault-tolerant scheduling and energy management for real-time embedded systems. Many of them have focused on dealing with transient faults through software redundancy, i.e., to reserve recovery jobs, whenever possible, for the jobs subject to transient faults. Zhu *et al.* [1] formulated the reliability as the probability of executing the real-time tasks/jobs successfully. Li *et al.* [22] introduced an adaptive scheme to minimize energy consumption under reliability requirement. Their work targeted the "frame-based" real-time systems only. For systems with more general real-time constraints, Zhao *et al.* [5] proposed an approach to reduce energy for periodical real-time tasks with reliability requirement quantified for each task individually. When considering shared resource synchronization, Zhang *et al.* [6] proposed a scheme to reduce energy consumption under reliability requirement. Most of the above works targeted real-time systems subject to transient faults only.

More recently, in order 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], and [23], online power management schemes applying dynamic voltage/freqency scaling (DVFS) in the primary processor and DPM in the spare processor were proposed. Moreover, in order to reduce energy consumption, Haque *et al.* [9] proposed to run the main tasks/jobs in the primary processor as soon as possible, while the backup tasks/jobs in the spare processor as late as possible such that once the main tasks/jobs are completed successfully, their corresponding backup tasks/jobs could be (partially) canceled. To enhance energy savings in both processors, in [24], a more advanced technique, named the *preference-oriented scheme*, was adopted which, in both the primary and backup processors, lets some tasks be scheduled as soon as possible, while the other ones be scheduled as late as possible. Their approach mainly implemented the preference-oriented scheduling from the task level in which all jobs belong to the same task have the same preference for execution. For standby-sparing systems with mixed criticality, advanced energy management schemes were proposed in [25]. Their approach tried to reduce energy through convex optimization in combination with power management heuristics based on joint DVFS and DPM schemes in both the primary and spare processors. When considering the chip thermal effect, peak-power-aware standby-sparing techniques utilizing energy management schemes were presented in [26]. Their approaches 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 FP scheduling policies, standby-sparing schemes based on the procrastination of the backup tasks were studied in [10]. In [11], more advanced FP standby-sparing techniques based on task-level preference-oriented scheduling schemes were explored with the purpose of reducing the energy further.

For multicore/multiprocessor systems, some works have also been done to reduce energy consumption. Zhou *et al.* [27] proposed an approach to minimize energy consumption for heterogeneous real-time multiprocessor systems under the thermal constraint. Chen and Thiele [28] proposed energy efficient task partitioning and platform synthesis methods for both DVFS and non-DVFS platforms. Neither of them has taken fault tolerance into consideration. Han *et al.* [29] explored fault-tolerant energy minimization for real-time systems on multiprocessor platforms using the checkpointing technique. Their approach can tolerate transient faults quite well. However, if a permanent fault happened, the checkpointing technique in [29] might not be able to deal with it effectively because, due to the real-time constraints, the failed part of the job might not be able to recovered successfully even in a different processor. Consequently, critical information on the failed job could be lost and the system might not be able to be restored timely. Note that all of the aforementioned existing works are mainly focused on *hard* real-time systems.

Different from the previous researches, the novelty of our proposed work in this article lies in the fact that we tried to combine the standby-sparing technique and (m, k)-deadlines to achieve better energy efficiency for real-time systems that can tolerate both transient and permanent faults.

III. Preliminaries

A. System Models

The real-time system considered in this article contains n independent periodic tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ scheduled according to the FP scheme. Without loss of generality, we assume that task τ_l has a lower priority than task τ_k if $l > k$. Each task contains an infinite sequence of periodically arriving instances called *jobs*. Task τ_i is characterized using five parameters, i.e., $(P_i, D_i, C_i, m_i, k_i)$. P_i, D_i $(\leq P_i)$, and C_i represent the period, the deadline, and the worst case execution time (WCET) for τ_i, respectively. A pair of integers, i.e., (m_i, k_i) $(0 < m_i \leq k_i)$, are used to represent the (m, k)-constraint for task τ_i that requires that among any k_i consecutive jobs, at least m_i jobs are executed successfully. The jth 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 standby-sparing system consists of two identical processors, which are denoted as primary processor and spare processor, respectively. For the purpose of tolerating permanent/transient faults, each mandatory job of a task τ_i has two duplicate copies running in the primary and the spare processors separately. Whenever a permanent fault is encountered in either processor, the other one will take over the whole system (to continue as normal). For convenience, we call each task τ_i *main task* and its corresponding copy running in the other processor *backup task*, denoted as τ_i'. The jth job of task τ_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 article, J_{ij}'s backup job, i.e., $\tilde{J}_{ij}$ might be different from J_{ij}', i.e., the job of τ_i' in the same time frame as J_{ij} because, as will be shown in later part of this article, J_{ij} and $\tilde{J}_{ij}$ can be shifted away from each other completely such that they might belong to different time frames.

B. Energy Model

The processor can be in one of the three states: 1) busy; 2) idle; and 3) *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 nonzero rising and falling time of the input and output signals. The dynamic power can be represented [30] as

$$P_{\text{dyn}} = C_L V^2 f. \tag{1}$$

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

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

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

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

When the processor is idle, it consumes the idle power (denoted as P_{idle}) whose major portion comes from the static power. When the processor is in the sleeping state, it consumes the sleeping power (denoted as P_{sleep}) which is assumed to be negligible. Note that although dynamic power can be reduced effectively by DVFS techniques, the efficiency of DVFS in reducing the overall energy is becoming seriously degraded with the dramatic increase in static power (mainly due to leakage) with the shrinking of IC technology size. With that in mind, in this article, we assume that the processors and the hardware platform used for standby sparing do not apply DVFS. As such, when the processors is busy, it always consumes P_{busy} at the maximal supply voltage $V_{\max}$. 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 $t_{sd} = \max([E_o/(P_{\text{idle}} - P_{\text{sleep}})], t_o)$. Correspondingly we call t_{sd} the *minimal shut-down interval*.

C. Fault Model

Similar to the standby-sparing systems in [9] and [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 that 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 [31] and the overhead for detection can be integrated into the job's execution time. Whenever a main job encounters transient fault(s), its backup job needs to be executed to completion.

IV. Motivations

Our goal is to reduce the overall energy consumption for standby-sparing systems under the (m, k) requirement. To assure the (m, k)-deadlines, a widely adopted strategy is to judiciously partition the jobs into *mandatory* jobs and *optional* jobs [32]. Two well-known partitioning strategies proposed are the *the deeply red pattern* (or *R*-pattern) [33] and *the evenly distributed pattern* (or *E*-pattern) [15]. According to the *R*-pattern, the pattern π_{ij} for job J_{ij}, i.e., the jth job of a task τ_i, is defined by (here, "1" represents the mandatory job and "0" represents the optional job)

$$\pi_{ij} = \begin{cases} \text{“1”}, & \text{if } 1 \leq j \bmod k_i \leq m_i \\ \text{“0”}, & \text{otherwise} \end{cases} \qquad j = 1, 2, 3, \ldots. \tag{4}$$

According to the E-pattern, the pattern π_{ij} for job J_{ij} is defined by

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

The mandatory/optional job partitioning according to (5) has the property that it helps to spread out the mandatory jobs *evenly* in each task along the time.

The necessary and sufficient condition for checking the schedulability of the fixed-priority task sets partitioned based on the E-pattern has been provided in [15]. The same rationale could also be applied for checking the schedulability of task sets partitioned based on the R-pattern as well.

From the above system models, to provide fault tolerance, all mandatory jobs based on the R-pattern or E-pattern 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 mandatory job in the primary processor be finished as soon as possible and its backup job in the spare processor be executed as late as possible such that once the main job is completed successfully, its backup job can be canceled immediately. To achieve this goal, Haque *et al.* [10] proposed to run the main tasks in the primary processor according to the regular FP scheme and the backup tasks on the spare processor according to the dual priority scheme. Their approach is based on the concept of “promotion time” (denoted as Y_i), calculated as follows:

$$Y_i = D_i - R_i \tag{6}$$

where R_i is the worst case response time of task τ_i.

By applying dual priority, each backup job from backup task τ_i' in the spare processor could be procrastinated by Y_i time units such that the overlapped executions between the main job and its backup job could be reduced, thereby saving energy. The energy reduction could be further boosted by adopting the preference-oriented scheduling scheme in [11]. Generally, their approach is quite efficient in reducing energy consumption for hard real-time systems. However, for soft real-time applications with (m, k)-deadlines, there still exist opportunities to reduce the energy further 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. This could be illustrated in the following example.

Given a task set of two tasks, i.e., $\tau_1 = (5, 4, 3, 2, 4)$ and $\tau_2 = (10, 10, 3, 1, 2)$, to be executed in a standby-sparing system. From (6), the promotion times Y_1 and Y_2 for tasks τ_1 and τ_2 are calculated as 1 and 1, respectively. By applying the preference-oriented approach in [11] to the mandatory jobs under the R-pattern, the main task τ_1 and backup task τ_2' will be scheduled in the primary processor (with τ_2' scheduled under dual priority) while main task τ_2 and backup task τ_1' will be scheduled in the spare processor (with τ_1' scheduled under dual priority). The schedules for them are shown in Fig. 1(a) and (b), respectively. As a result, the total active energy consumption within the hyper period [0, 20] is 15 units.[1]

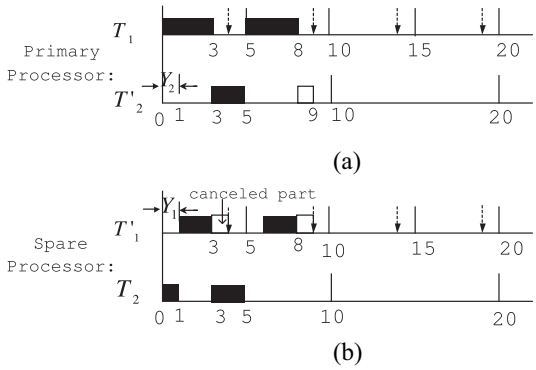


Fig. 1. (a) Schedule for the main task τ_1 and backup task τ_2' in the primary processor under the preference-oriented scheme [11]. (b) Schedule for the backup task τ_1' and main task τ_2 in the spare processor under the preference-oriented scheme [11].

Note that in the above example, there are still extensive overlapped execution times between the mandatory jobs and their corresponding backup jobs, incurring significant energy consumption. The main issue is that with the above approach, only the mandatory jobs (and their backup jobs) under the static R-patterns were executed. Note that the static patterns such as the R-patterns defined in (4) only contain a “minimal set” of mandatory jobs in them, no more no less, which just meet the given (m, k)-constraint. Therefore, to ensure fault tolerance, each mandatory job needs to have a backup job reserved for it in the other processor, no exception. As a result, each mandatory main job needs to be executed concurrently with its backup job in the same time frame, which could result in significant overlapped execution times between them due to the time constraint. However, if we look into the optional jobs and consider executing them adaptively, there could be more chance to save energy. The reason is: since the optional jobs are not required in meeting the given (m, k)-constraint, they do not need backup jobs reserved for them. Moreover, once any optional job was completed successfully, it would be counted as a valid job that could contribute to satisfying the given (m, k)-constraint as well, which means some mandatory job (together with its backup job) in the near future might not need to be executed anymore. In this way, by exploring the possibility of executing the optional jobs and adjusting the pattern dynamically, there could be more opportunities in saving energy. This could be shown in Fig. 2. As seen in Fig. 2(a) and (b), under dynamic patterns, the first job of task τ_2 is determined and scheduled as an optional job, denoted as O_{21}, instead of mandatory because it can still tolerate one more deadline missing.[2] Once O_{21} is executed and completed successfully, its next mandatary job, i.e., J_{22}, can be demoted

[1]Note that for easy of presentation, in all examples in this article, 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.

[2]Although O_{11} is also determined as an optional job, we chose to execute O_{21} first because, starting from O_{11}, task τ_1 can still tolerate two deadlines missings; therefore, it is regarded as more flexible (less urgent) than O_{21}.

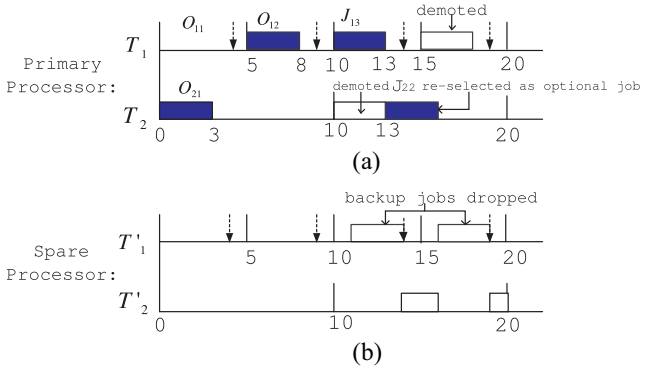


Fig. 2. (a) Schedule for the main tasks τ_1 and τ_2 in the primary processor based on the greedy execution of the optional jobs. (b) Backup jobs [for the original mandatory jobs in (a)] in the spare processor are dropped.

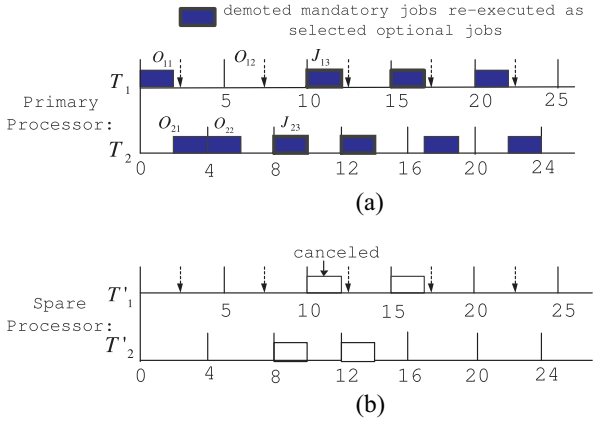


Fig. 3. (a) Schedule for the main tasks $\tau_1 = (5, 2.5, 2, 2, 4)$ and $\tau_2 = (4, 4, 2, 2, 4)$ in the primary processor under the greedy execution of the optional jobs. (b) Schedule for the backup jobs in the spare processor.

to optional and the backup job for it can simply be dropped to save energy. After O_{21} was completed (at time $t = 3$), since O_{11} did not have enough time space to be finished before its deadline, to save energy, O_{11} will not be invoked at all. Instead, its next optional job, i.e., O_{12} will be invoked at time $t = 5$, following the same rationale as O_{21}. Note that in the schedule in Fig. 2(a), although some mandatory jobs, such as J_{13} and J_{22} had been demoted to optional,[3] they were reselected (as optional jobs) for execution again to help demote/drop more mandatory/backup jobs in the future. As a result the total active energy consumption within the hyper period [0, 20] is reduced to 12 units, which is 20% lower than that in Fig. 1.

It is not hard to see that in Fig. 2 the fault tolerance capability of the standby-sparing system is preserved as whenever some optional job(s) failed, the next mandatory job (and the backup job for it) can still be invoked and executed timely.

From the above example, we can see that by executing optional jobs and adjusting the job patterns dynamically, there is great potential for saving energy because the optional jobs do not need backup jobs. Moreover, the successful completion of the optional jobs can help demote/drop some mandatory jobs and their backup jobs in the future. Additionally, when those mandatory jobs are demoted, their time budget could be utilized to execute more optional jobs. However, although seeming reasonable, there could still be problems with it. For example, due to the "greedy" manner [4] in which the optional jobs are executed, it might execute an excessive number of optional jobs for some systems with modest workload, which could affect the overall energy efficiency adversely. Even we limit the execution of the optional jobs to be in one processor (e.g., the primary processor) only, the same problem might still exit. This could be illustrated with another example as followed.

Consider another task set of two tasks, i.e., $\tau_1 = (5, 2.5, 2, 2, 4)$ and $\tau_2 = (4, 4, 2, 2, 4)$. Fig. 3 shows the schedule based on the greedy manner. As can be seen, for task τ_1, the execution of optional job O_{11} caused mandatory job J_{13}

[3]To different such kinds of demoted mandatory jobs from those original optional jobs, we still use their original job names in the figure when it does not cause any confusion.

[4]Here "greedy manner" means the optional jobs are scheduled whenever possible to do so.

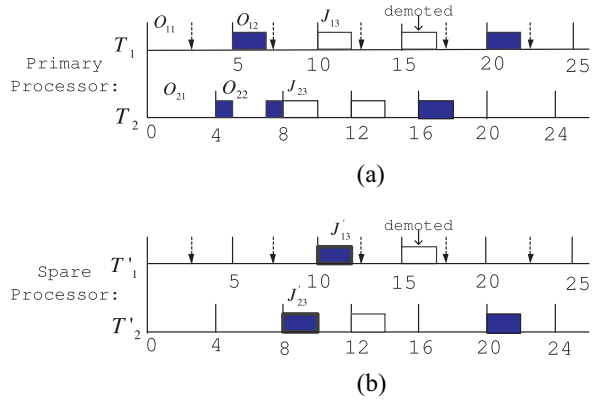


Fig. 4. (a) Schedule for the main tasks $\tau_1 = (5, 2.5, 2, 2, 4)$ and $\tau_2 = (4, 4, 2, 2, 4)$ in the primary processor based on the selective execution of the optional jobs. (b) Schedule for the backup tasks τ_1' and τ_2' in the spare processor based on the selective execution of the optional jobs.

(and its backup job) to be demoted/dropped. But, later on, J_{13} was reselected for execution as an optional job. Following the same rationale, four jobs in total were executed for task τ_1 before time $t = 25$. The situation for task τ_2 is similar. As a result, the total active energy consumption under this schedule is 20 units.

However, if we follow a different schedule as shown in Fig. 4, the energy efficiency can be improved further. As can be seen, in this case, for both tasks τ_1 and τ_2, we only scheduled those optional jobs that could tolerate just one more deadline missing (such as O_{12} and O_{22}) while skipping the other optional jobs (e.g., O_{11} and O_{21}). Moreover, to make the workload of the optional jobs distribute more evenly between two processors, we let the selected optional jobs of each task be executed in the primary processor and the spare processor alternatively. For example, for O_{12} and O_{22} , we let them be executed in the primary processor. Once O_{12} and O_{22} were finished successfully, the flexibility degrees (FDs) of the future jobs would be updated correspondingly, based on which the jobs could be reselected for execution again (e.g., J_{13}' and J_{23}', which would be executed in the spare processor). Following the same principle, the total active energy consumption before

time $t = 25$ was reduced to 14 units, which is 30% lower than that in Fig. 3.

As shown in the above example, it is not necessarily most energy efficient to execute the optional jobs in a greedy manner while executing them selectively might be more promising in saving energy. Moreover, to better utilize the system computing power, in the latter case, we can let the selected optional jobs of each task be executed in both the primary and the spare processors alternatively, which could help distribute their workloads more evenly in the two processors such that the selected optional jobs could have better chance to be scheduled successfully. Following these rationales, in the next section, we will propose a new approach which, instead of executing all optional jobs in one processor greedily, will execute them in both the primary and spare processors in a selective manner.

V. Approach Based on *R*-Pattern

In this section, we will present our new approach based on selective execution of the optional jobs according to the *R*-pattern. The following definition would be very useful in presenting our algorithm.

Definition 1: The FD of a job J_i, denoted as $FD(J_i)$, is defined as the number of consecutive deadline missings that task τ_i (which J_i belongs to) can still tolerate starting from J_i.

Based on the concept of FD, our selective approach works according to the following principles: 1) the optional jobs could be executed in either the primary or the spare processor, but only the optional jobs with FD of 1 will be selected for execution and 2) the eligible optional jobs are to be executed either under the main tasks in the primary processor or under the backup tasks in the spare processor, but not both for the same optional job (as an optional job does not have backup job). Regarding that, to make the workload of the eligible optional jobs distribute more evenly in two processors, we let the selected optional jobs from the same task be executed in the primary processor and in the spare processor alternatively, just as shown in the schedule in Fig. 4.

The salient part of our selective approach is presented in Algorithm 1.

As shown in Algorithm 1, for both the primary and the spare processors, two job ready queues are maintained for each of them: 1) a mandatory job queue (MJQ) and 2) an optional job queue (OJQ). Upon arrival, the current job of task τ_i (denoted as J_i) is determined as mandatory job or optional job based on its FD. It is determined as mandatory only if its FD is 0 and as optional otherwise. Note that since all mandatory jobs must have backup jobs for them, we let the mandatory jobs of all main tasks be put in the MJQ of the primary processor while their corresponding backup jobs be put in the MJQ of the spare processor. Unlike the mandatory jobs, the optional jobs do not have backup jobs for them. So, only the optional jobs with FD of 1 are selected as eligible jobs (other optional jobs are skipped). Moreover, the selected optional jobs of each task are put into the OJQ of the primary processor and the OJQ of the spare processor alternatively. The jobs in MJQ always have higher priorities than those in OJQ.

Algorithm 1 Selective Approach

1: **For either the primary processor or the spare processor:**
2: **if** *MJQ* is not empty **then**
3: If in primary processor, run jobs in *MJQ* under FP scheme; Whenever a main job is completed successfully, cancel its backup job in the other processor immediately.
4: If in spare processor, run jobs in *MJQ* under FP scheme with job arrival times revised according to Equation (7);
5: **else if** *OJQ* is not empty **then**
6: Select J_i in OJQ and run it following the FP scheme;
7: **if** J_i is executed successfully **then**
8: Updated the flexibility degree of the next job of the same task;
9: **end if**
10: **else**
11: t_{cur} = the current time;
12: t'_a = the earliest release time of all jobs in MJQ;
13: **if** $(t'_a - t_{\text{cur}}) > T_{be}$ **then**
14: Shut down the processor and set the wake-up timer to be $(t'_a - t_{\text{cur}})$;
15: **end if**
16: **end if**

Note that during runtime, once an optional job is completed successfully, it will be counted as a valid job and the FD of the next job should be updated correspondingly (lines 7 and 8).

In addition, to facilitate saving energy for running the backup jobs in the spare processor when necessary, the executions of all backup jobs in the spare processor should be postponed as late as possible. To achieve this goal, some offline analysis could be done based on the following definitions.

Definition 2: Time t is called the *postponed release time*, denoted as $\tilde{r}_i$, of a backup job J'_i in the spare processor and is calculated as

$$\tilde{r}_i = r_i + \theta_i \tag{7}$$

where θ_i is calculated with (9).

Definition 3: Time t is called a J_{ij}*-inspecting point* for job J_{ij}, denoted as $\mathcal{IP}(J_{ij})$, if $t = d_{ij}$ or $t \in \{\tilde{r}_{kl} \mid k < i \text{ and } r_{ij} < \tilde{r}_{kl} < d_{ij}\}$, where $\tilde{r}_{kl}$ is the postponed release time of job J_{kl} calculated in (7).

Definition 4: The *job release postponement interval*, denoted as θ_{ij}, for any backup job J'_{ij} of task τ'_i is defined as

$$\theta_{ij} = \max\left\{\left(\bar{t} - \left(c_{ij} + \sum_{d_{kl} > r_{ij}, \tilde{r}_{kl} < \bar{t}}^{k<i} c_{kl}\right) - r_{ij}\right) \mid \bar{t} \in \mathcal{IP}\left(J'_{ij}\right)\right\} \tag{8}$$

where $\tilde{r}_{kl}$ is the postponed release time of job J_{kl} calculated in (7).

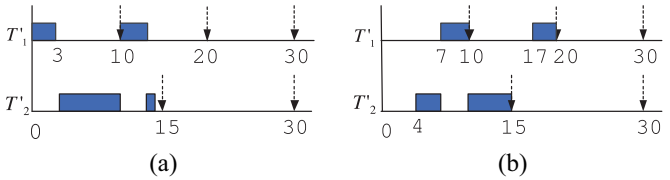


Fig. 5. (a) Original schedule of the backup jobs for the main tasks $\tau_1 = (10, 10, 3, 2, 3)$ and $\tau_2 = (15, 15, 8, 1, 2)$. (b) Schedule of the backup jobs based on the postponed release times calculated according to (7).

Definition 5: The *task release postponement interval*, denoted as θ_i, for any ask τ_i' is defined as

$$\theta_i = \max\left\{\min\left\{\theta_{ij} \mid j \leq \frac{\text{LCM}_{q\leq i}(k_q P_q)}{P_i}\right\},\ Y_i\right\} \tag{9}$$

where Y_i is the promotion time of τ_i calculated based on (6).

The calculation of θ_i can be done offline based on the static *R*-pattern. Note that since the postponed release times of the higher priority jobs will be used as the inspecting points for the lower priority jobs, the release postponement intervals for the backup tasks should be calculated in descending order of the priority levels. Each time when $\min\{\theta_{ij} \mid j \leq [(\text{LCM}_{q\leq i}(k_q P_q))/P_i]\}$ for level i is calculated, we can compare it with Y_i and choose the maximum between them as θ_i for level i. In this way, we can guarantee that θ_i is never less than Y_i. After that the release time of all backup jobs of task τ_i' should be revised based on (7) before advancing to the next priority level.

As an example of calculating the postponed release time $\tilde{r}_i$ and the release postponement interval θ_i, let us consider a task set of two tasks, i.e., $\tau_1 = (10, 10, 3, 2, 3)$ and $\tau_2 = (15, 15, 8, 1, 2)$, the original schedule of the backup jobs in the spare processor with nonpostponed release time is shown in Fig. 5(a). According to (6), $Y_1 = 7$ and $Y_2 = 1$. To calculate $\tilde{r}_{11}$, i.e., the postponed release time of the first backup job in τ_1' (represented as J_{11}'), there is only one inspecting point for it, i.e., $\bar{t} = 10$. Based on (8), $\theta_{11} = 10 - 3 - 0 = 7$. Similarly, θ_{12} can be calculated as 7 as well. So, according to (9), $\theta_1 = \max\{7,\ Y_1\} = 7$. After that, the release time of all backup jobs of task τ_1' should be revised according to (7), as shown in Fig. 5(b). Next, to calculate θ_{21} for the first job of τ_2' (represented as J_{21}'), according to Definition 3, there are two inspecting points for it, i.e., 15 and 7, based on its deadline and the postponed release times of the jobs in τ_1'. Then, according to (8), $\theta_{21} = \max\{15 - (8 + 3) - 0, 7 - (8 + 0) - 0\} = 4$. Since for this particular example, there is only one backup job in τ_2' within its hyper period ($\text{LCM}_{q\leq 2}(k_q P_q) = 30$), $\theta_2 = \max\{4,\ Y_2\} = 4$. After that the release times of all backup jobs in τ_2' are postponed by 4 time units according to (7). The schedule based on the postponed release times of all jobs within the first hyperperiod is shown in Fig. 5(b). It is not hard to see that under this postponed schedule all backup jobs can meet their deadlines. Note that for this particular example, the release postponement interval calculated for task τ_2', i.e., θ_2, is much larger than the promotion time of τ_2' calculated according to (6), i.e., $Y_2 = 1$.

The complexity of Algorithm 1 mainly comes from scheduling the optional jobs in the primary and the spare processors. Since at anytime there are at most n optional jobs in the OJQ, its complexity is $O(n)$. Moreover, to ensure that the (m, k)-deadlines be satisfied, we have the following theorem.

Theorem 1: Let task set $\mathcal{T}$ be scheduled with Algorithm 1. The (m, k)-deadlines for $\mathcal{T}$ can be ensured if $\mathcal{T}$ is schedulable under the *R*-pattern.

Proof: The correctness of the release postponement interval θ_i for each backup task τ_i' (and its individual backup jobs) is guaranteed by (8) and (9) because according to (8) and the definition of J_{ij}-inspecting point in Section V, the completion time of any backup job will never go beyond its deadline.

The worst case scenario of Algorithm 1 happens when at certain time point t, in both the primary and spare processors, the optional jobs of each task are either not selected for execution or not completed successfully. Then, the next m_i jobs of each task τ_i should be designated as mandatory jobs consecutively in order to meet the (m, k)-constraint. Let r_e be the earliest arrival time of all upcoming mandatory jobs after time t. If we shift left all other tasks such that the arrival time of the next upcoming mandatory job of each task coincides with r_e, it is easy to see that after such kind of shifting the task set will become harder to be schedulable than the original one as the work demand that is required to be finished before any job deadline after t will not be decreased. On the other hand, it is easy to see that the situation of the shifted task set after t is the same as when all tasks are released synchronously at time 0 under the *R*-pattern.

The situation for the backup jobs in the spare processor is the same if we replace the release time(s) above with the postponed release time(s) of the backup jobs. The conclusion of Theorem 1 follows. ■

VI. Approach Based on *E*-Pattern

Although the approach in Section VII-A is quite efficient in reducing the energy consumption for task sets partitioned based on the *R*-pattern, the schedulability of the *R*-pattern is not as good as the *E*-pattern [34]. On the other hand, it is not possible to quantize a utilization threshold for task sets partitioned under any (m, k)-pattern because the utilization of the nonschedulable task set with (m, k)-constraint could be arbitrarily low. In the following, we will formulate that into a lemma.

Lemma 1: The utilization of the nonschedulable task set with a (m, k)-constraint could be arbitrarily low.

Proof: Without loss of generality, consider a task set containing only one task (P_i, D_i, C_i, m_i, k_i) in which $C_i > D_i$ and $m_i << k_i$. Then, obviously this task set is not schedulable. Meanwhile, since $m_i << k_i$, its total utilization, i.e., $\sum_i [(m_i C_i)/(k_i P_i)]$, could be arbitrarily low. ■

It is not hard to see that the approach in Section V for the *R*-pattern can be applied to task sets partitioned based on the *E*-pattern as well. However, when doing so, its energy efficiency might not be as good as for task sets partitioned based on the *R*-pattern. This could be illustrated in the following example.

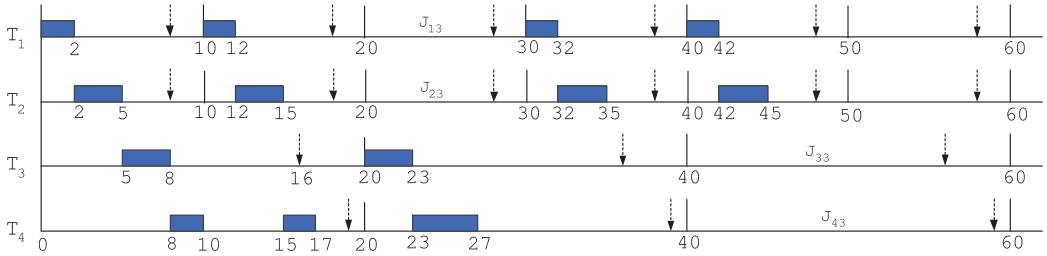


Fig. 6. Schedule for task set $\{\tau_1 = (10, 8, 2, 4, 6),\ \tau_2 = (10, 8, 3, 4, 6),\ \tau_3 = (20, 16, 3, 2, 3),\ \tau_4 = (20, 19, 4, 2, 3)\}$ based on the original E-pattern.

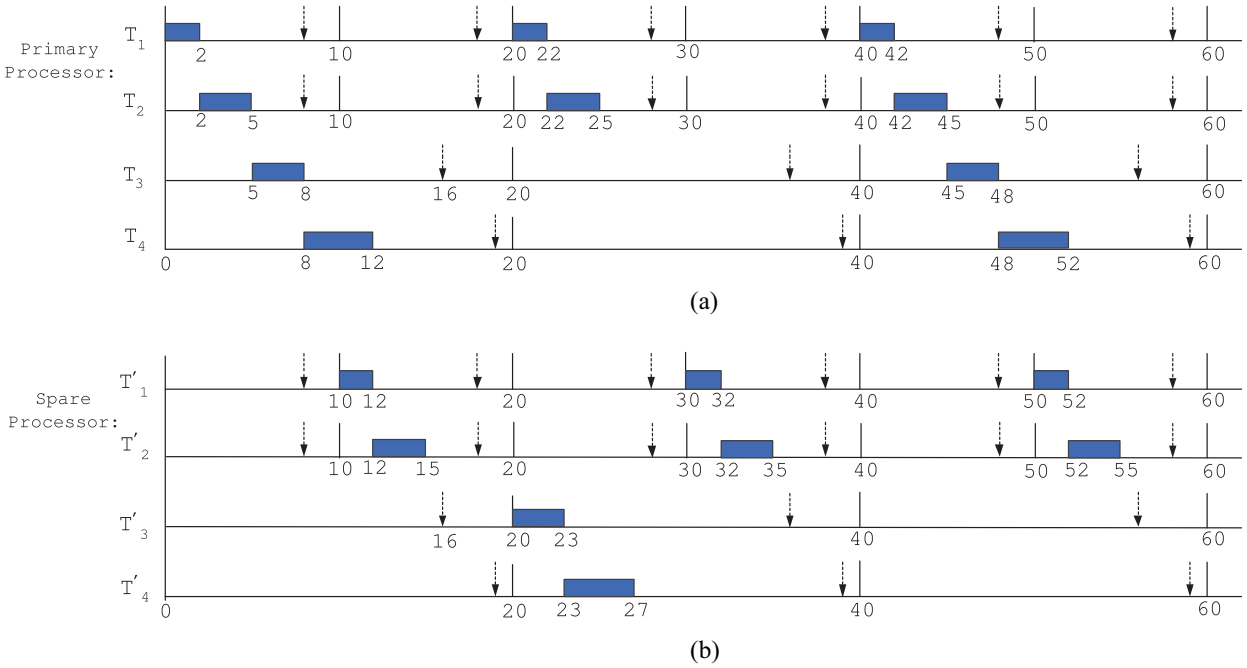


Fig. 7. (a) Schedule for the main tasks $\tau_1 = (10, 8, 2, 4, 6),\ \tau_2 = (10, 8, 3, 4, 6),\ \tau_3 = (20, 16, 3, 2, 3)$, and $\tau_4 = (20, 19, 4, 2, 3)$ in the primary processor based on the selective approach in Algorithm 1 based on the E-pattern. (b) Schedule for the backup tasks τ_1', τ_2', τ_3', and τ_4' in the spare processor based on the selective approach in Algorithm 1 based on the E-pattern.

Consider another task set of four tasks: i.e., $\tau_1 = (10, 8, 2, 4, 6)$, $\tau_2 = (10, 8, 3, 4, 6)$, $\tau_3 = (20, 16, 3, 2, 3)$ and $\tau_4 = (20, 19, 4, 2, 3)$. If we assume there is no fault occurring during the first hyper period, the schedule based on the original E-pattern (for either the main tasks in the primary processor or the backup tasks in the spare processor) is shown in Fig. 6, which will result in a total active energy consumption of 68 units if no energy management is applied. It is not hard to see that all optional jobs in it (such as J_{13}, J_{23}, J_{33}, and J_{43}) have FD of 1. If we apply the selective approach in Section V to it, as shown in Fig. 7, all optional jobs of each task will be executed either in the primary processor or the spare processor alternatively and none of them got chance to be skipped. As a result, the total active energy consumption for executing the jobs in the primary processor [Fig. 7 (a)] and in the spare processor [Fig. 7(b)] will be 51 units.

However, if we adopt a different way of scheduling the task set, we can achieve still better energy efficiency. The *central idea* is to execute the mandatory main/backup jobs determined under the E-pattern according to the *job-level preference-oriented scheme*, which could execute each individual mandatory main/backup job either as soon as possible or as late as possible based on the *triple priority scheme*.[5] In order to do so, we need to enhance the calculation of the promotion time of each mandatory job under the E-pattern such that the mandatory jobs could be procrastinated as late as possible, following the steps.

1) *Step 1:* For all tasks, shift their mandatory job patterns using the approach in [32] such that the mandatory workload from each individual task could be shifted away from one another as far as possible.
2) *Step 2:* Based on the shifted mandatory job pattern generated in step 1, apply the approach in [36] to calculate the optimal promotion time for each mandatory job in each task τ_i.

Note that in the above step (step 2), in order to procrastinate the mandatory jobs to the furthest time, instead of using the promotion time calculated by (6), here we choose to use the optimal promotion time for each mandatory job. As shown in [36], for periodic task sets, the value calculated in (6) is only a lower bound for the promotion time and, usually, the optimal promotion time could be much larger, especially under

[5]The triple priority scheme is a variation of the dual priority scheme in [35].

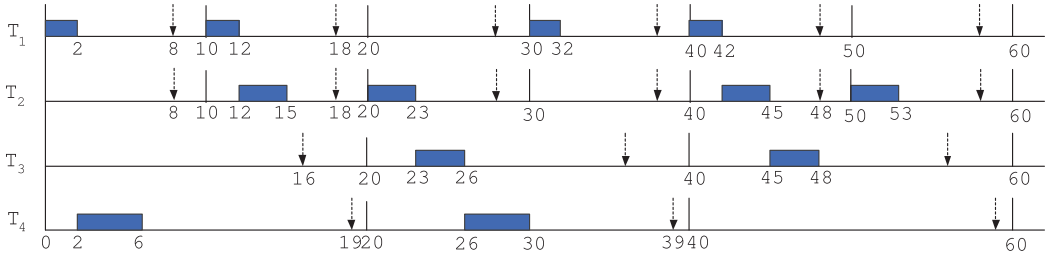


Fig. 8. Schedule for the task set $\tau_1 = (10, 8, 2, 4, 6)$, $\tau_2 = (10, 8, 3, 4, 6)$, $\tau_3 = (20, 16, 3, 2, 3)$, and $\tau_4 = (20, 19, 4, 2, 3)$ based on the shifted E-pattern using the approach in [32].

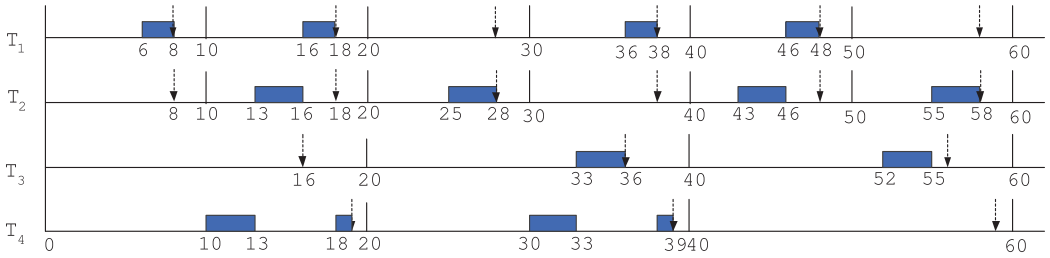


Fig. 9. Schedule for the mandatory jobs in Fig. 8 after applying the optimal promotion times on them.

the (m, k)-constraint. Interested readers are referred to [36] for more details on the calculation of the optimal promotion time for each mandatory job. The above steps (steps 1 and 2) could also be illustrated using the schedules in Figs. 8 and 9, respectively. As shown in Fig. 8, for the same example task set in Fig. 6, we first try to shift the E-pattern of each task using the approach in [32] such that the mandatory workloads from all tasks could be shifted away from one another to the maximal extent and the interference from higher priority tasks to lower priority tasks could be minimized. Next, based on the shifted patterns in Fig. 8, we apply the approach in [36] to calculate the optimal promotion times for the mandatory jobs, which results in $Y^*_{11} = Y^*_{12} = Y^*_{14} = Y^*_{15} = 6$, $Y^*_{22} = Y^*_{25} = 3$, $Y^*_{23} = Y^*_{26} = 5$, $Y^*_{32} = 13$, $Y^*_{33} = 12$, and $Y^*_{41} = Y^*_{42} = 10$, respectively. The schedule for all mandatory jobs based on the optimal promotion times is shown in Fig. 9.

With the shifted patterns and the optimal promotion times calculated above, we can reduce the energy by applying the *job-level preference-oriented* scheme in executing the mandatory main/backup jobs. The main idea is: for each mandatory main/backup job J_i, it could be executed only when its current preference mode (denoted as J_i^P) has been assigned a valid execution mode, i.e., S (representing as soon as possible execution) or L (representing as late as possible execution). Based on it the jobs will be scheduled according to the following principles.

1) For each mandatory main job or its backup job, it will be executed following the *triple priority scheme*, i.e., at anytime, it could reside in only one of the three priority levels in its own processor: 1) lower; 2) middle; and 3) upper levels. Upon arrival, each mandatory main/backup job will be released at the lower priority level first, with its corresponding preference mode initialized as "NULL." Whenever a mandatory main job or its backup job, whichever first, gets chance to be dispatched in its own processor, it will be switched to the middle-priority level of its processor immediately and its preference mode will be updated as S (as soon as possible mode). At the same time, its corresponding job in the other processor will be procrastinated to its optimal promotion time and switched to the upper priority level with its preference mode updated as L (as late as possible mode). Moreover, whenever a mandatory main/backup job reaches its optimal promotion time, it must be switched to the upper priority level immediately.
2) The jobs in the upper priority level can always preempt the jobs in the middle-priority level and the jobs in the middle-priority level can always preempt the jobs in the lower priority level. For all jobs in the same priority level, they will be executed following their original fixed priority assignment.
3) At any time, between the mandatory main job and its corresponding backup job, at most one of them could be switched to (and executed in) the middle-priority level. Moreover, if one of them has been switched to the middle-priority level, the other one must be switched to the upper priority level immediately and procrastinated to its optimal promotion time.
4) Whenever slack time becomes available, the jobs under the as soon as possible preference mode (S) should try to reclaim the slack time for execution (to facilitate its early completion) while the jobs under the as late as possible preference mode (L) should never reclaim the slack time for execution. Instead, it should try to utilize the slack time to implement dynamic procrastination such that it could be delayed further.

To help explain the above rationale, we still use the same example task set in Fig. 6 based on their output in Fig. 9. The schedule is demonstrated in Fig. 10.

As shown in Fig. 10 (a) and (b), at time 0, when J_{11} and its backup job J'_{11} are released, both of them are put in the lower priority level first. Since there is no pending job

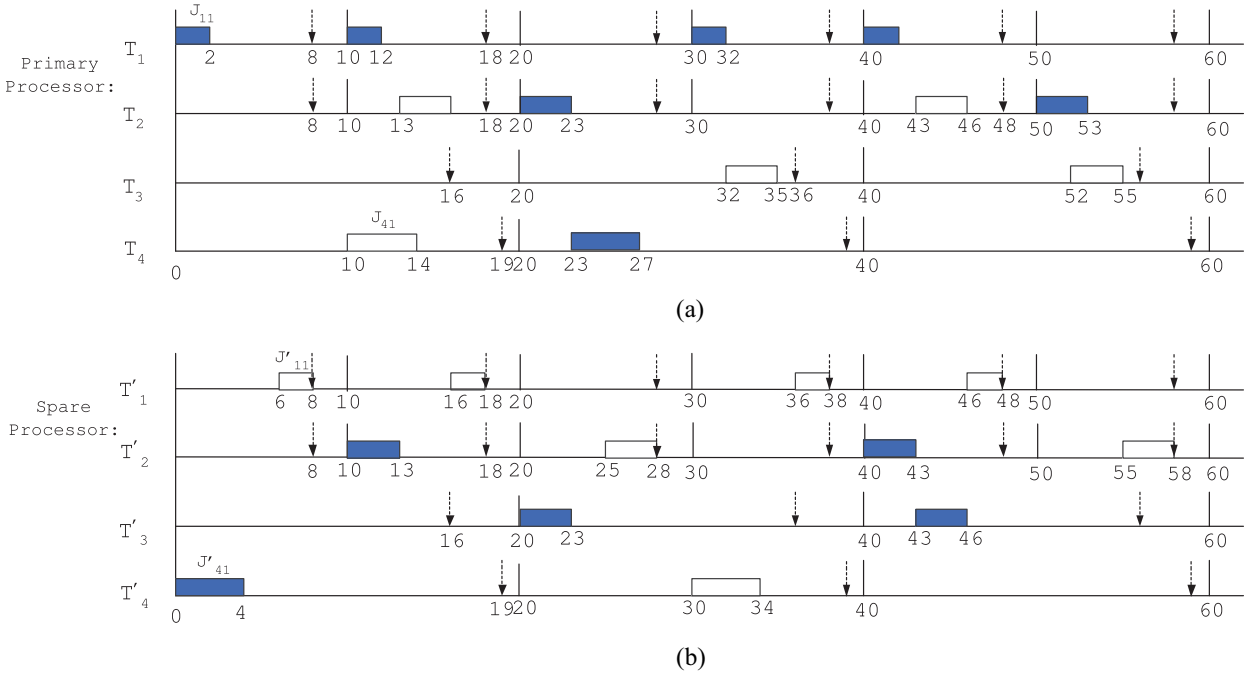


Fig. 10. (a) Schedule for the main tasks $\tau_1 = (10, 8, 2, 4, 6)$, $\tau_2 = (10, 8, 3, 4, 6)$, $\tau_3 = (20, 16, 3, 2, 3)$, and $\tau_4 = (20, 19, 4, 2, 3)$ in the primary processor using the job-level dynamic preference-oriented approach (Algorithm 2) based on the E-pattern. (b) Schedule for the backup tasks τ'_1, τ'_2, τ'_3, and τ'_4 in the spare processor using the job-level preference-oriented approach (Algorithm 2) based on the E-pattern.

with higher priority than J_{11} in the primary processor, J_{11} got chance to be dispatched immediately and was switched to the middle-priority level, while J'_{11} was procrastinated to its optimal promotion time $(0 + Y^*_{11}) = 6$ and switched to upper priority level from there. Meanwhile, at time 0, both J_{41} and its backup job J'_{41} are released in the lower priority level first. But since in the primary processor job J_{41} has lower priority than job J_{11}, it could not be dispatched at time 0. On the other hand, in the spare processor job J'_{41} got chance to be dispatched because job J'_{11} has been procrastinated to time 6. Therefore, J'_{41} would be switched to the middle priority level while J_{41} would be procrastinated to its optimal promotion time $(0 + Y^*_{41}) = 10$ (and switched to upper priority level from there). When J_{11} is completed successfully at time 2, J'_{11} in the spare processor could be canceled and became slack time from there. Similarly, when J'_{41} is completed successfully at time 4, J_{41} in the primary processor could be canceled and became slack time from there. Following the same rationale, the complete schedule for the remaining mandatory main/backup jobs in the primary and the spare processors are shown in Fig. 10 (a) and (b), respectively. It is not hard to see that under this schedule, the total active energy consumption in both processors could be reduced to 34 units, which is 34% lower than that in Fig. 7.

To formalize the above procedure, the main steps are shown in Algorithm 2.

As shown in Algorithm 2, our online scheduling algorithm is based on the triple priority scheme. During runtime, in both the primary and the spare processors, three mandatory job ready queues, i.e., MJQ^u, MJQ^m, and MJQ^l are maintained, which correspond to the three priority levels under the triple priority scheme, i.e., lower, middle, and upper priority levels. Upon arrival, a mandatory main job (and its backup job in the other processor) is first put in the lower mandatory queue MJQ^l of its processor. Note that both of the mandatory main/backup jobs have their preference modes initialized as "NULL" upon arrival. Thereafter, when either of them, whichever first, gets chance to be dispatched, it will be switched to the middle priority queue MJQ^m with its preference mode reset to be S [representing as soon as possible execution, for example, job J_{11} in Fig. 10(a)] and its corresponding job in the other processor will be switched to the upper priority queue MJQ^u with its preference mode reset to be L [representing as late as possible execution, for example, job J'_{11} in Fig. 10(b)]. For all jobs in MJQ^l and MJQ^m, whenever their optimal promotion times are reached, they will be switched to the upper mandatory queue MJQ^u while their current preference mode will be preserved.

Algorithm 2 Job-Level Preference-Oriented Scheme

1: **For either the primary processor or the spare processor:**
2: **if** MJQ^u is not empty **then**
3: Scheduling the jobs in MJQ^u based on Algorithm 3;
4: **else if** MJQ^m is not empty **then**
5: Scheduling the jobs in MJQ^m based on Algorithm 4;
6: **else if** MJQ^l is not empty **then**
7: Scheduling the jobs in MJQ^l based on Algorithm 5;
8: **else**
9: Repeat lines 11-15 of Algorithm 1;
10: **end if**

Algorithm 3 Scheduling the Jobs in the Upper Priority Level

1: **For either the primary processor or the spare processor:**
2: Pick $J_i \in \text{MJQ}^u$ with the smallest index;
3: **if** $J_i^P ==$ "S" **then**
4: // J_i's is in as soon as possible preference mode;
5: Execute J_i following the FP scheme as early as possible;
6: **if** any slack time $ST_i(t)$ with higher priority than J_i is available **then**
7: Reclaim the slack time to execute J_i as early as possible;
8: **end if**
9: **else**
10: // J_i's is under as late as possible preference mode "L";
11: Delay the starting time of J_i's execution to $(t_{\text{cur}} + ST_i(t))$;
12: Execute J_i following the FP scheme;
13: **end if**
14: **if** the execution of job J_i is successful **then**
15: Cancel its corresponding job in the other processor and add the residue time budget to the slack queue $\mathcal{ST}$;
16: **end if**

Algorithm 4 Scheduling the Jobs in the Middle-Priority Level

1: **For either the primary processor or the spare processor:**
2: Pick $J_i \in \text{MJQ}^m$ with the smallest index;
3: Execute J_i following the FP scheme as early as possible;
4: **if** any slack time $ST_i(t)$ with higher priority than J_i is available **then**
5: Reclaim the slack time to execute J_i as early as possible;
6: **end if**
7: **if** the execution of job J_i is successful **then**
8: Cancel its corresponding job in the other processor and add the residue time budget to the slack queue $\mathcal{ST}$;
9: **end if**

During runtime, in both the primary and the spare processors, a slack queue $\mathcal{ST}$ needs to be maintained to keep track of the slack time(s) from (partially) canceled job(s).

Whenever a mandatory main/backup job is completed successfully, its corresponding job in the other processor should be canceled whose residue time budget will become slack time (line 15 in Algorithm 3 and line 8 in Algorithm 4). The slack time(s) will be inserted into $\mathcal{ST}$ based on their priorities. Upon the starting execution of a job J_i at time t, the slack time from $\mathcal{ST}$ with priorities higher than or equal to J_i will be stored in a variable $ST_i(t)$. If J_i's category is S, $ST_i(t)$ should be reclaimed to execute J_i as soon as possible (lines 6–8 in Algorithm 3). Otherwise, if J_i's category is L, $ST_i(t)$ should be used to implement dynamic procrastination on J_i, which can delay J_i to $(t_{\text{cur}} + ST_i(t))$ (line 11 in Algorithm 3). When no job is pending for execution, the system should be put into sleep mode if the idle time is larger than the break even time (line 9 in Algorithm 2). Note that even when the system is in sleep (or idle) mode, the slack times stored in $\mathcal{ST}$ should still be consumed based on their sorted sequence in $\mathcal{ST}$.

Algorithm 5 Scheduling the Jobs in the Lower Priority Level

1: **For either the primary processor or the spare processor:**
2: Pick $J_i \in \text{MJQ}^l$ with the smallest index;
3: Let $\tilde{J}_i$ be J_i's corresponding job in the other processor;
4: **if** $\tilde{J}_i$ is also in MJQ^l of its own processor **then**
5: **if** the MJQ^u and MJQ^m in both processors are empty **then**
6: Randomly pick out a job between J_i and $\tilde{J}_i$ to be promoted to MJQ^m;
7: Reset the preference mode of the picked job to be "S" while procrastinating its corresponding job in the other processor to its optimal promotion time and resetting its preference mode to be "L";
8: **else**
9: Pick out the job in the processor with empty MJQ^u or MJQ^m to be promoted to MJQ^m;
10: Repeat line 7;
11: **end if**
12: **end if**

The complexity of Algorithm 2 mainly comes from scheduling the jobs in the three different MJQs and switching preference mode for each of them. Since at anytime, there are at most n mandatory jobs in each MJQ and for each mandatory job there are at most two preference modes (plus an initialized "NULL" mode), its complexity is $O(n)$.

VII. Performance Evaluation

In this section, we compared our approach with other previous related approaches with simulations. Note that due to the difference in the schedulability of the R-pattern and E-pattern, we conducted two groups of simulations separately: one for R-pattern-based schemes and one for E-pattern-based schemes.

A. Evaluation of R-Pattern-Based Schemes

In this part, we evaluated the energy performance of R-pattern-based schemes. Three different approaches were studied. In the first approach, the task sets were statically partitioned with R-patterns, and the mandatory jobs in the primary and the spare processors were executed concurrently without procrastination. We refer to this approach as MKSS_{ST} and used its results as the reference. The second approach (MKSS_{DP}) also determined the mandatory jobs based on the static R-patterns and the mandatory jobs were scheduled with the task-level preference-oriented scheme based on dual priority, similar to that used in [11] (but without applying DVFS). The third approach ($\text{MKSS}^R_{\text{selective}}$) is our selective approach proposed in Section V based on the selective execution of the optional jobs in both the primary and the spare processors.

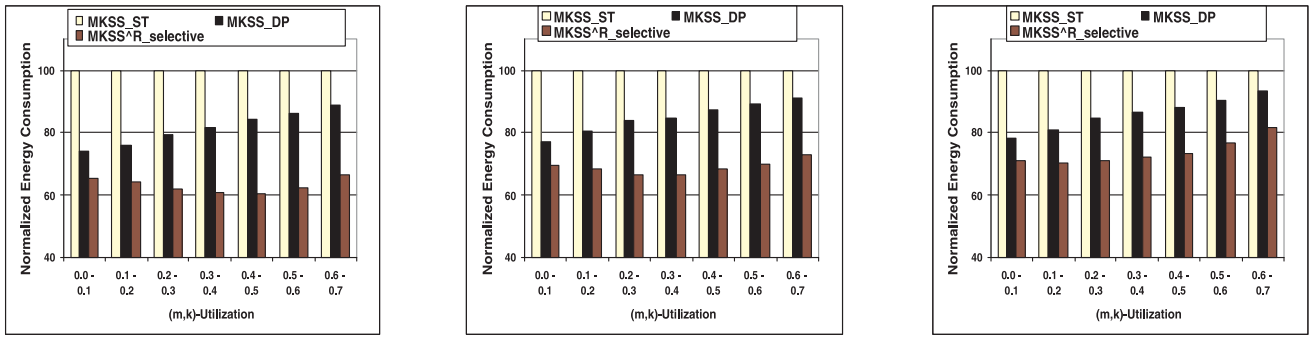


Fig. 11. Energy comparisons for R-pattern-based schemes subject to: (a) no fault, (b) permanent fault, and (c) permanent and transient faults.

The processor model used in our simulations is based on the Free-scale PowerQUICC III integrated Communications Processor MPC8536E [37], similar to the one used in [38]. According to the data sheet in [37], the typical power consumption of MPC8536E running under the maximal frequency is 4.7 W (with a core frequency of 1500 MHz and core voltage of 1.1 V). The idle power P_{idle} is about 0.6 W. Since the transition overheads are not mentioned in the data sheet, we assumed the shut-down/wake-up time overhead $t_o = 1$ ms and energy overhead $E_o = 0.6$ mJ. Therefore, the minimal shut-down interval t_{sd} will be calculated as 1 ms.

The periodic task sets in our experiments consisted of five to ten tasks with the periods randomly chosen in the range of [5, 50] ms. m_i and k_i for the (m, k)-deadlines were also randomly generated such that k_i was uniformly distributed between 2 and 20, and $0 < m_i < k_i$. The WCET of a task was assumed to be uniformly distributed and the total (m, k)-utilization, i.e., $\sum_i [(m_i C_i)/(k_i P_i)]$, 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 three sets of tests.

In the first set, we checked the energy performance when no fault occurred within the hyperperiod. The results are shown in Fig. 11(a).

From Fig. 11(a), one can immediately see that by adopting dynamic patterns, $\text{MKSS}^R_{\text{selective}}$ can achieve much better energy efficiency than the others adopting static patterns, i.e., MKSS_{ST} and MKSS_{DP}, in all utilization intervals. The maximal energy reduction by $\text{MKSS}^R_{\text{selective}}$ over MKSS_{DP} can be around 26%. The main reason is that in this scenario, by executing the optional jobs, $\text{MKSS}^R_{\text{selective}}$ can help drop duplicate executions of the mandatory/backup jobs in two processors significantly. Moreover, with the adaptive optional job selection strategy, i.e., by only choosing optional jobs with FD of 1 for execution, $\text{MKSS}^R_{\text{selective}}$ can avoid executing excessive number of the optional jobs. Additionally, by letting the selected optional jobs be executed in two different processors alternatively, $\text{MKSS}^R_{\text{selective}}$ can help distribute the workloads of the optional jobs in two processor evenly. Finally, by letting the backup jobs be delayed with the postponed release times, $\text{MKSS}^R_{\text{selective}}$ can accommodate larger pools of eligible optional jobs for selection, which also gives more chance for the optional jobs to be selected and scheduled successfully, therefore minimizing the necessity of running mandatory jobs effectively.

In the second set, we assumed the system is subject to permanent fault only which could occur at most once. The results are shown in Fig. 11(b).

As seen in Fig. 11(b), the energy reduction by our new approaches, i.e., $\text{MKSS}^R_{\text{selective}}$ subject to permanent fault is similar to the case when no fault ever occurred. Compared to MKSS_{DP}, the energy saving by $\text{MKSS}^R_{\text{selective}}$ can be up to 20% for the same reasons as above.

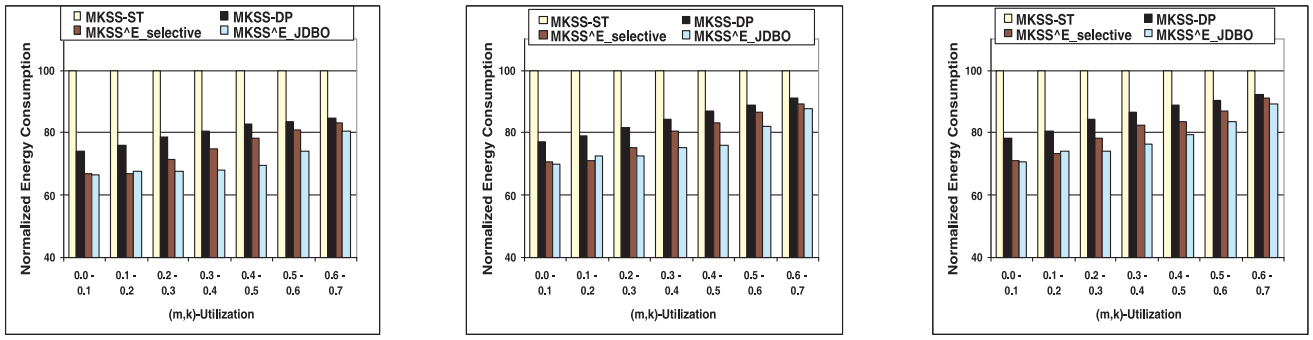
In the third set, we assumed the system could be subject to both permanent fault and transient faults. The transient fault model is similar to that used in [1] by assuming the Poisson distribution with an average fault rate of 10^{-6}. The results were shown in Fig. 11(c).

As seen, the energy saving by our new approach, i.e., $\text{MKSS}^R_{\text{selective}}$ in this scenario is similar to that in the previous cases. The maximal energy reduction by $\text{MKSS}^R_{\text{selective}}$ over MKSS_{DP} can be up to 15%, thanks to the adaptive executions of the optional jobs under the dynamic pattern adjustment.

B. Evaluation of E-Pattern-Based Schemes

In this part, we evaluated the energy performance of the E-pattern-based schemes. We studied four different approaches. In the first approach, the task sets were statically partitioned with E-patterns, and the mandatory jobs in the primary and the spare processors were executed concurrently without procrastination. We still refer to this approach as MKSS_{ST} and used its results as the reference. The second approach (MKSS_{DP}) also determined the mandatory jobs based on the static E-patterns and the mandatory jobs were scheduled with the task-level preference-oriented scheme based on dual priority, similar to that used in [11] (but without applying DVFS). In the third approach, for comparison purpose, we tried to apply our selective approach proposed in Section V to the task sets partitioned based on the E-pattern as well. We refer to it as $\text{MKSS}^E_{\text{selective}}$. The fourth approach $\text{MKSS}^E_{\text{JDPO}}$ is our job-level preference-oriented approach proposed in Section VI.

The periodic task sets are generated in the same way as in Section VII-A but we allow $m_i = k_i$. We also assume the processor shut-down break even time $T_{be} = 1$ ms.



(a) (b) (c)

Fig. 12. Energy comparisons for E-pattern-based schemes subject to: (a) no fault, (b) permanent fault, and (c) permanent and transient faults.

In the first set, we checked the energy performance when no fault occurred within the hyper period. The results are shown in Fig. 12(a).

From Fig. 12(a), one can see that when the mandatory workload of the task set is relatively low (e.g., when the (m, k)-utilization is less than 0.3), our approach adopting dynamic patterns, i.e., $\text{MKSS}^E_{\text{selective}}$ can still achieve obviously better energy efficiency than the previous approaches, i.e., MKSS_{ST} and MKSS_{DP}. This is mainly due to the fact that when the (m, k)-utilization is low, even under the E-pattern, quite a lot optional jobs could still have FD larger than 1 and $\text{MKSS}^E_{\text{selective}}$ might not execute excessive number of optional jobs. Therefore, it is still quite efficient in saving energy during these utilization intervals. On the other hand, it is also easy to see that the energy reduction achievable by $\text{MKSS}^E_{\text{selective}}$ over the previous approaches is not the same as the case for task sets partitioned based on the R-pattern as shown in Section VII-A. This is mainly because, under the E-pattern, due to the even distribution of the mandatory/optional job patterns, $\text{MKSS}^E_{\text{selective}}$ might not be able to skip as many optional jobs as it did under the R-pattern. As a result, the energy efficiency of the selective approach for task sets partitioned with E-pattern might not be as aggressive as that for task sets partitioned with R-pattern. Meanwhile, it is easy to see that under this scenario the energy consumption of $\text{MKSS}^E_{\text{JDPO}}$ could be quite close to or slightly higher than that of $\text{MKSS}^E_{\text{selective}}$. However, when (m, k)-utilization is greater than or equal to 0.3, the energy consumption of $\text{MKSS}^E_{\text{selective}}$ was increasing very fast and became much higher than that of $\text{MKSS}^E_{\text{JDPO}}$. This is mainly because, under this scenario, due to the even distribution of the job patterns under the E-pattern, most optional jobs in it had FD of 1 and nearly all of them could be selected for execution by $\text{MKSS}^E_{\text{selective}}$, incurring much higher energy consumption due to the excessive number of optional jobs executed. Different from that, $\text{MKSS}^E_{\text{JDPO}}$ can save energy more efficiently in this case. The maximal energy reduction by $\text{MKSS}^E_{\text{JDPO}}$ over MKSS_{DP} and $\text{MKSS}^E_{\text{selective}}$ can be around 14% and 12%, respectively. The main reason is that under this scenario, by executing the mandatory jobs only, $\text{MKSS}^E_{\text{JDPO}}$ can avoid executing too many jobs. Moreover, by adopting the job-level preference-oriented approach under the triple priority scheme, the workloads of the mandatory/backup jobs in the primary and the spare processors could be shifted away in a more adaptive manner, resulting in more aggressive energy savings.

In the second set, we assumed the system is subject to permanent fault only which could occur at most once. The results are shown in Fig. 12(b).

As seen in Fig. 12(b), the energy savings by our new approach, i.e., $\text{MKSS}^E_{\text{JDPO}}$ subject to permanent fault is similar to the case when no fault ever occurred. When the (m, k)-utilization is not extremely low, compared with MKSS_{DP} and $\text{MKSS}^E_{\text{selective}}$, the energy reduction by $\text{MKSS}^E_{\text{JDPO}}$ can be up to 12% and 10%, respectively, for the same reasons as stated above.

In the third set, we assumed the system could be subject to both permanent fault and transient faults. The transient fault model is the same as used in Section VII-A. The results were shown in Fig. 12(c).

As seen, the energy saving by our new approach, i.e., $\text{MKSS}^E_{\text{JDPO}}$ in this scenario is similar to that in the previous cases. The maximal energy reduction by $\text{MKSS}^E_{\text{JDPO}}$ over MKSS_{DP} and $\text{MKSS}^E_{\text{selective}}$ can be up to 10% and 9%, respectively, for the same reasons as stated above.

VIII. Conclusion

Energy consumption, QoS, and fault tolerance are among the most critical factors in the real-time systems design. In this article, we presented two novel FP scheduling schemes for reducing energy consumption in standby-spare systems while assuring (m, k)-deadlines and fault tolerance: one for task sets partitioned with deeply red pattern and one for task sets partitioned with evenly distributed pattern. As shown, the proposed approaches outperformed the previous research significantly in energy conservation while assuring the (m, k)-deadlines and fault tolerance for real time applications under FP assignment.

References

[1] D. Zhu, R. Melhem, and D. Mosse, "The effects of energy management on reliability in real-time embedded systems," in *Proc. ICCAD*, 2004, pp. 35–40.

[2] P. Bose, J. Rivers, C.-K. Hu, J. Srinivasan, and S. V. Adve, "RAMP: A model for reliability aware microprocessor design," Dept. Comput. Sci., IBM Res., Armonk, NY, USA, Rep. RC23048(W0312-122), 2003.

[3] D. Zhu, "Reliability-aware dynamic energy management in dependable embedded real-time systems," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 2, p. 26, Jan. 2011.

[4] Z. Li, L. Wang, S. Li, S. Ren, and G. Quan, "Reliability guaranteed energy-aware frame-based task set execution strategy for hard real-time systems," *J. Syst. Softw.*, vol. 86, no. 12, pp. 3060–3070, Dec. 2013.
[5] B. Zhao, H. Aydin, and D. Zhu, "Energy management under general task-level reliability constraints," in *Proc. IEEE 18th Real Time Embedded Technol. Appl. Symp.*, Washington, DC, USA, 2012, pp. 285–294.
[6] Y.-W. Zhang, H.-Z. Zhang, and C. Wang, "Reliability-aware low energy scheduling in real time systems with shared resources," *Microprocess. Microsyst.*, vol. 52, pp. 312–324, Jul. 2017.
[7] A. Taherin, M. Salehi, and A. Ejlali, "Reliability-aware energy management in mixed-criticality systems," *IEEE Trans. Sustain. Comput.*, vol. 3, no. 3, pp. 195–208, Jul.–Sep. 2018.
[8] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "Low-energy standby-sparing for hard real-time systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 3, pp. 329–342, Mar. 2012.
[9] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware standby-sparing technique for periodic real-time applications," in *Proc. ICCD*, 2011, pp. 190–197.
[10] M. A. Haque, H. Aydin, and D. Zhu, "Energy-aware standby-sparing for fixed-priority real-time task sets," *Sustain. Comput. Inf. Syst.*, vol. 6, pp. 81–93, Jun. 2015.
[11] R. Begam, Q. Xia, D. Zhu, and H. Aydin, "Preference-oriented fixed-priority scheduling for periodic real-time tasks," *J. Syst. Archit.*, vol. 69, pp. 1–14, Sep. 2016.
[12] Y.-W. Zhang, "Energy-aware mixed partitioning scheduling in standby-sparing systems," *Comput. Stand. Interfaces*, vol. 61, pp. 129–136, Jan. 2019.
[13] M. Shirazi, M. Kargahi, and L. Thiele, "Performance maximization of energy-variable self-powered (m, k)-firm real-time systems," *Real-Time Syst.*, vol. 56, pp. 64–111, Feb. 2020.
[14] M. Hamdaoui and P. Ramanathan, "A dynamic priority assignment technique for streams with (m,k)-firm deadlines," *IEEE Trans. Comput.*, vol. 44, no. 12, pp. 1443–1451, Dec. 1995.
[15] P. Ramanathan, "Overload management in real-time control applications using (m,k)-firm guarantee," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 6, pp. 549–559, Jun. 1999.
[16] M. Chetto, "Graceful overload management in firm real-time systems," *J. Inf. Technol. Softw. Eng.*, vol. 5, no. 3, pp. 1–3, 2015.
[17] O. Gettings, S. Quinton, and R. I. Davis, "Mixed criticality systems with weakly-hard constraints," in *Proc. 23rd Int. Conf. Real Time Netw. Syst.*, 2015, pp. 237–246.
[18] G. V. D. Brüggen, K.-H. Chen, W.-H. Huang, and J.-J. Chen, "Systems with dynamic real-time guarantees in uncertain and faulty execution environments," in *Proc. IEEE Real-Time Syst. Symp. (RTSS)*, Nov. 2016, pp. 303–314.
[19] Y. Sun and M. D. Natale, "Weakly hard schedulability analysis for fixed priority scheduling of periodic real-time tasks," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, p. 171, Sep. 2017. [Online]. Available: http://doi.acm.org/10.1145/3126497
[20] T. A. AlEnawy and H. Aydin, "Energy-constrained scheduling for weakly-hard real-time systems," in *Proc. RTSS*, 2005, pp. 376–385.
[21] H. Kooti, N. Dang, D. Mishra, and E. Bozorgzadeh, "Energy budget management for energy harvesting embedded systems," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, Aug. 2012, pp. 320–329.
[22] Z. Li, S. Ren, and G. Quan, "Energy minimization for reliability-guaranteed real-time applications using DVFS and checkpointing techniques," *J. Syst. Archit.*, vol. 61, no. 2, pp. 71–81, Feb. 2015.
[23] A. Ejlali, B. M. Al-Hashimi, and P. Eles, "A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems," in *Proc. CODES+ISSS*, Oct. 2009, pp. 193–202.
[24] Y. Guo, H. Su, D. Zhu, and H. Aydin, "Preference-oriented real-time scheduling and its application in fault-tolerant systems," *J. Syst. Archit.*, vol. 61, no. 2, pp. 127–139, Feb. 2015.
[25] S. Safari, S. Hessabi, and G. Ershadi, "LESS-MICS: A low energy standby-sparing scheme for mixed-criticality systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4601–4610, Dec. 2020.
[26] M. Ansari, A. Yeganeh-Khaksar, S. Safari, and A. Ejlali, "Peak-power-aware energy management for periodic real-time applications," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 4, pp. 779–788, Apr. 2020.
[27] J. Zhou, T. Wei, M. Chen, J. Yan, S. Hu, and Y. Ma, "Thermal-aware task scheduling for energy minimization in heterogeneous real-time MPSOC systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 8, pp. 1269–1282, Aug. 2016.
[28] J.-J. Chen and L. Thiele, "Task partitioning and platform synthesis for energy efficiency," in *Proc. 15th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.* Aug. 2009, pp. 393–402.
[29] Q. Han, M. Fan, and G. Quan, "Energy minimization for fault tolerant real-time applications on multiprocessor platforms using checkpointing," in *Proc. ISLPED*, 2013, pp. 76–81.
[30] L. Niu and G. Quan, "Reducing both dynamic and leakage energy consumption for hard real-time systems," in *Proc. CASES*, Sep. 2004, pp. 140–148.
[31] D. K. Pradhan, Ed., *Fault-tolerant Computing: Theory and Techniques*, vol. 2. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.
[32] G. Quan and X. Hu, "Enhanced fixed-priority scheduling with (m,k)-firm guarantee," in *Proc. RTSS*, 2000, pp. 79–88.
[33] G. Koren and D. Shasha, "Skip-Over: Algorithms and complexity for overloaded systems that allow skips," in *Proc. RTSS*, 1995, p. 110.
[34] L. Niu and G. Quan, "Energy minimization for real-time systems with (m,k)-guarantee," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 7, pp. 717–729, Jul. 2006.
[35] R. I. Davis and A. J. Wellings, "Dual priority scheduling," in *Proc. RTSS*, 1995, pp. 100–109.
[36] G. Bernat and A. Burns, "Combining (n,m)-hard deadlines and dual priority scheduling," in *Proc. RTSS*, Dec. 1997, pp. 46–57.
[37] *MPC8536E PowerQUICC III Integrated Processor Hardware Specifications, Revision 7*, document MPC8536EEC, Freescale Semicond., Austin, TX, USA, Jul. 2015. [Online]. Available: https://www.nxp.com/docs/en/data-sheet/MPC8536EEC.pdf
[38] M. A. Awan and S. M. Petters, "Race-to-halt energy saving strategies," *J. Syst. Archit.*, vol. 60, no. 10, pp. 796–815, 2014.

Linwei Niu (Member, IEEE) received the B.S. degree in computer science and technology from Peking University, Beijing, China, in 1998, the M.S. degree in computer science from the State University of New York at Stony Brook, Stony Brook, NY, USA, in 2001, and the Ph.D. degree in computer science and engineering from the University of South Carolina, Columbia, SC, USA, in 2006.

He is currently an Assistant Professor with the Department of Electrical Engineering and Computer Science, Howard University, Washington, DC, USA. His research interests include power-aware design for embedded systems, design automation, real-time scheduling, and software/hardware co-design.

Dr. Niu is a member of the IEEE Computer Society.

Dakai Zhu (Senior Member, IEEE) received the B.E. degree in computer science and engineering from Xi'an Jiaotong University, Xi'an, China, in 1996, the M.E. degree in computer science and technology from Tsinghua University, Beijing, China, in 1999, and the Ph.D. degree in computer science from the University of Pittsburgh, Pittsburgh, PA, USA, in 2004.

He is currently a Professor with the Department of Computer Science, University of Texas at San Antonio, San Antonio, TX, USA. His research interests include real-time systems, fault tolerance, and power-aware computing.

Prof. Zhu was a recipient of the U.S. National Science Foundation Faculty Early Career Development (CAREER) Award in 2010. He is a senior member of the IEEE Computer Society.