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

Linwei Niu
Department of Math and Computer Science
West Virginia State University
Institute, WV, U.S.A.
lniu@wvstateu.edu

Dakai Zhu
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, U.S.A.
dakai.zhu@utsa.edu

***Abstract*—For real-time computing systems, energy efficiency, Quality of Service, and fault tolerance are among the major design concerns. In this work, we study the problem of reliable and energy-aware fixed-priority (m,k)-deadlines enforcement with standby-sparing. The standby-sparing systems adopt a primary processor and a spare processor to provide fault tolerance for both permanent and transient faults. In order to reduce energy consumption for such kind of systems, we proposed a novel scheduling scheme under the QoS constraint of (m,k)-deadlines. The evaluation results demonstrate that our proposed approach 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, standby-sparing, (m,k)-deadlines, fixed-priority scheduling**

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 plenty of works (e.g. [3], [4]) have been reported in conserving energy for fault-tolerant real-time systems. Many of them has focused on dealing with transient faults. A widely adopted strategy is to use software redundancy, *i.e.*, to reserve recovery jobs, whenever possible, for the jobs subject to transient faults. For mission critical applications such as nuclear plant control systems, permanent faults need to be dealt with by all means to avoid system failure. Otherwise catastrophical consequences could occur. To address this issue, solutions adopting hardware redundancy are required. Among them the *standby-sparing* technique has recently gained much attention in research community [5]–[9]. Generally, the standby-sparing makes use of the redundancy of processing units in multicore/multiprocessor systems. More specifically, a standby-sparing system consists of two processors, a primary one and a spare one, executing in parallel. For each real-time job executed in the primary processor, there is a corresponding backup job reserved for it in the spare processor [7]. As such, whenever a permanent fault occurs to the primary or the spare processor, the other one can still continue without causing system failure. Moreover, it is not hard to see that the backup tasks/jobs in the spare processor can also help tolerate transient faults for their corresponding main tasks/jobs in the primary processor.

In a standby-sparing system, the execution of the main jobs in the primary processor and their corresponding backup jobs in the spare processor might need to be overlapped with each other. Thus the total energy consumption could be quite considerable. Regarding that, some recent works have been reported to reduce energy (e.g. [5]–[7], [9]). 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 [7], [8], approaches based on dual priority scheme were proposed for standby-sparing fixed-priority real-time systems. Their works are mainly focused on *hard* real-time systems.

In most real-time applications such as multimedia or time-critical communication systems, occasional deadline missings are acceptable so long as the user perceived quality of service (QoS) can be assured at certain levels. For such kind 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 [10]. To ensure the (m,k)-deadlines, Ramanathan *et al.* [11] proposed to partition the jobs into *mandatory* ones and *optional* ones. The mandatory ones must be completed successfully whereas the other ones could be optionally executed when necessary.

In this paper, we study the problem of reliable and energy-aware fixed-priority (m,k)-deadlines enforcement with standby-sparing. 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 rest of the paper is organized as follows. Section II presents the preliminaries. Section III presents the motivations.

Demo Version, http://www.verydoc.com and http://www.verypdf.com

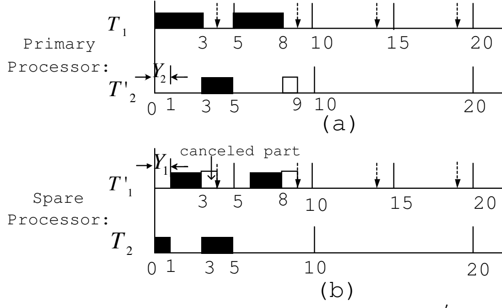

Fig. 1. (a) The schedule for the main task τ_1 and backup task τ_2' in the primary processor under the preference oriented scheme [8]; (b) The schedule for the backup task τ_1' and main task τ_2 in the spare processor under the preference oriented scheme [8].

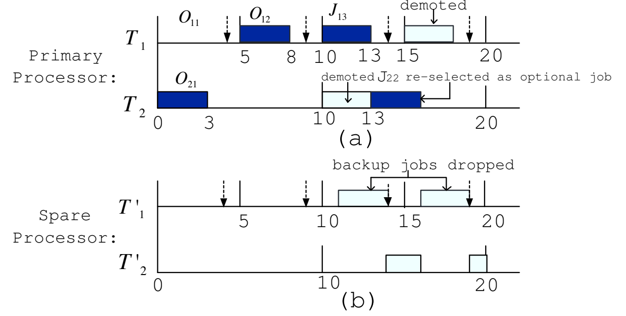

Fig. 2. (a) The schedule for the main tasks τ_1 and τ_2 in the primary processor based on the greedy execution of the optional jobs; (b) The backup jobs (for the original mandatory jobs in (a)) in the spare processor are dropped.

Section IV presents our proposed approach. In Section V and Section VI, we present our evaluation results and conclusions.

II. Preliminaries

A. System models

The real-time system considered in this paper contains n independent periodic tasks, $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$, scheduled according to the fixed-priority (FP) scheme. Without loss of generality, we assume that τ_j has a lower priority than τ_i if $j > i$. 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) $(0 < m_i < k_i)$, 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 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 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 in the same time frame. Whenever a permanent fault in encountered in either processor, the other one will take over the whole system (to continue as normal). For convenience, we call each task τ_i (or mandatory job J_{ij}) running in the primary processor *main task/job* and its corresponding copy running in the spare processor *backup task/job*, denoted as τ_i' (or J_{ij}').

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

B. Fault Model

Similar to the standby-sparing systems in [6], [7], the system we considered can tolerate both permanent and transient faults. With the redundancy of the processing units, our system can tolerate at least one permanent fault in the primary or the spare processor. For transient faults which can occur anytime during the task execution, we assume they can be detected at the end of a job's execution using sanity (or consistency) checks [12] 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.

III. 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 [13]. A well-known partitioning method is called the *deeply red pattern* (or R-pattern) [14]. According to R-pattern, the pattern π_{ij} for job J_{ij}, *i.e.*, the jth job of a task τ_i, is defined by:

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

where "1" represents the mandatory job and "0" represents the optional job.

From the above system models, to provide fault tolerance, all mandatory jobs based on R-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, in [7] Mohammad *et. al* proposed to run the main tasks in the primary processor according to regular FP scheme and the backup tasks on the spare processor according

Demo Version, http://www.verydoc.com and http://www.verypdf.com

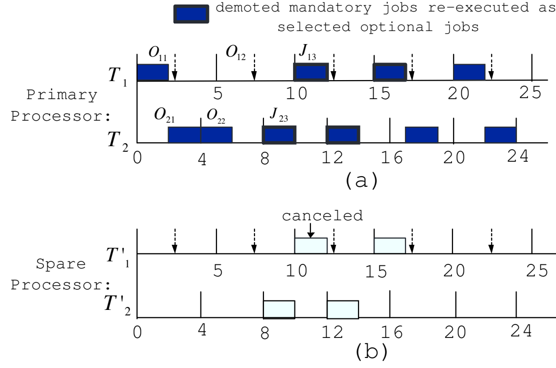

Fig. 3. (a) The schedule for the main tasks $\tau_1 = (5, 2.5, 2, 2, 4)$, $\tau_2 = (4, 4, 2, 2, 4)$ in the primary processor under the greedy execution of the optional jobs; (b) The schedule for the backup jobs in the spare processor.

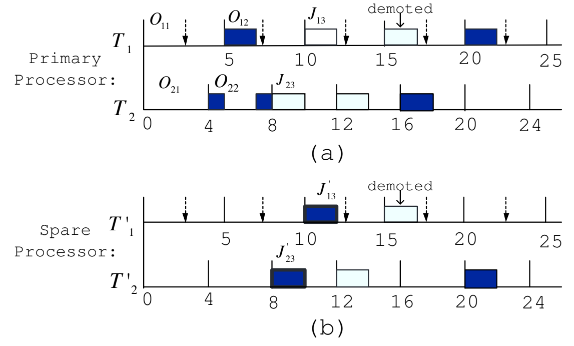

Fig. 4. (a) The schedule for the main tasks $\tau_1 = (5, 2.5, 2, 2, 4)$, $\tau_2 = (4, 4, 2, 2, 4)$ in the primary processor based on the selective execution of the optional jobs; (b) The schedule for the backup tasks τ_1' and τ_2' in the spare processor based on the selective execution of the optional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

to the dual priority scheme. Their approach is based on the concept of "promotion time" (denoted as Y_i), calculated as followed:

$$Y_i = D_i - R_i \tag{2}$$

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 [8]. 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.*, $\tau_1 = (5, 4, 3, 2, 4)$, $\tau_2 = (10, 10, 3, 1, 2)$, to be executed in a standby-sparing system. From equation (2), 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 [8] to the mandatory jobs, 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 Figure 1(a) and (b), respectively. As a result, the total active energy consumption within the hyper period [0,20] is 15 units.

Note that in the above example, there are still significant overlapped time between the executions of all mandatory jobs and their corresponding backup jobs, incurring much more energy consumption. On the other hand, due to the existence of the optional jobs, we can explore the possibility of executing them adaptively and adjusting the pattern dynamically, thereby saving more energy, which is shown in Figure 2. As seen in Figure 2(a) and (b), under dynamic patterns, the first job of task τ_2 is determined and scheduled as an optional job, represented as O_{21}, instead of mandatory because it can still tolerate one more deadline missing[1]. Once O_{21} is executed and completed successfully, its next mandatory job, *i.e.*, J_{22}, can be demoted 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 Figure 2(a), although some mandatory jobs such as J_{13} and J_{22} had been demoted to optional, they were re-selected (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 Figure 1.

It is not hard to see that in Figure 2 the fault tolerance capability of the standby-sparing system is preserved as whenever some 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 energy saving because for optional jobs, no backup jobs need to executed for them and the successful completion of the optional jobs can help demote/drop some mandatory jobs and their backup jobs. Moreover, when those mandatory jobs are demoted, their time budget could be utilized to execute more optional jobs. However, although seems reasonable, there could still be problems with it. For example, due to the "greedy" manner 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 the primary processor only, this problem might still not be avoided effectively. This could be illustrated with another example as followed.

Consider another task set of two tasks, *i.e.*, $\tau_1 = (5, 2.5, 2, 2, 4)$, $\tau_2 = (4, 4, 2, 2, 4)$. Figure 3 shows the schedule based on the greedy approach. As can be seen, for task τ_1,

[1] 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 is regarded as more flexible (less urgent) than O_{21}.

Demo Version, http://www.verydoc.com and http://www.verypdf.com

the execution of optional job O_{11} caused mandatory job J_{13} (and its backup job) to be demoted/dropped. But later on, J_{13} was re-selected 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 Figure 4, the energy efficiency can be improved. As can be seen, in this case for both tasks τ_1 and τ_2, we only schedule their optional jobs which can tolerate just one more deadline missing (such as O_{12} and O_{22}) while skipping the other optional jobs (for example, O_{11} and O_{21}). Moreover, to make the workload of the optional jobs distribute more evenly in 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} are finished, the flexibility degrees of the future jobs will be updated correspondingly, based on which the jobs could be selected for execution again (for example, J_{13} and J'_{23}, which will be executed in the spare processor). The total active energy consumption before time $t = 25$ is reduced to 14 units, which is 30% lower than that in Figure 3.

As shown in the above example, the greedy approach is not necessarily most energy efficient while executing the optional jobs selectively is 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 workload more evenly in two processors such that the selected optional jobs could have better chance to be scheduled successfully. Following these rationales, in 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 way.

IV. Our Approach

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

Definition 1: The ***flexibility degree*** 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 flexibility degree, our selective approach works according to the following principles: (i) The optional jobs could be executed in either the primary or the spare processor, but only the optional jobs with flexibility degree of 1 will be selected for execution; (ii) The eligible optional jobs 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 in the schedule in Figure 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: a mandatory job queue (MJQ) and 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 flexibility degree. It is determined as mandatory only if its flexibility degree 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 mandatory job queue (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 flexibility degree 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 The selective approach (*Selective-App*)

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 (3);
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** $(t'_a - t_{cur}) > T_{be}$ **then**
14: Shut down the processor and set the wake-up timer to be $(t'_a - t_{cur})$;
15: **end if**
16: **end if**

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

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 off-line 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

Demo Version, http://www.verydoc.com and http://www.verypdf.com

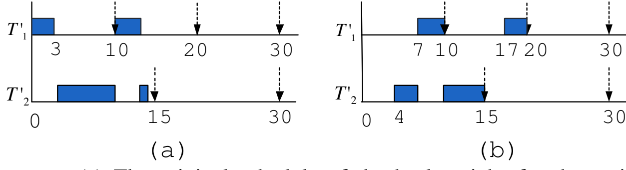

Fig. 5. (a) The original schedule of the backup jobs for the main tasks $\tau_1 = (10, 10, 3, 2, 3)$, $\tau_2 = (15, 15, 8, 1, 2)$; (b) The schedule of the backup jobs based on the postponed release times calculated according to equation (3).

is calculated as:

$$\tilde{r}_i = r_i + \theta_i \tag{3}$$

where θ_i is calculated with Equation (5).

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 $t \in \{\tilde{r}_{kl} \mid k < i \ and \ r_{ij} < \tilde{r}_{kl} < d_{ij}\}$.

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

$$\theta_{ij} = max\{(\bar{t} - (c_{ij} + \sum_{d_{kl} > r_{ij}, \tilde{r}_{kl} < \bar{t}}^{k<i} c_{kl}) - r_{ij}) \mid \bar{t} \in \mathcal{IP}(J'_{ij})\} \tag{4}$$

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

$$\theta_i = min\{\theta_{ij} \mid j \leq \frac{LCM_{q\leq i}(k_q P_q)}{P_i}\} \tag{5}$$

The calculation of θ_i can be done off-line based on 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 θ_i is calculated, the release time of all backup jobs of task τ'_i should be revised based on equation (3) before advancing to the next priority level. When all θ_i are calculated, if for any task τ'_i, θ_i is less than R_i, we can always set θ_i to be R_i safely.

As an example of calculating the postponed release time $\tilde{r}_i$ and the release postponement interval θ_i, let's consider a task set of two tasks, *i.e.*, $\tau_1 = (10, 10, 3, 2, 3)$, $\tau_2 = (15, 15, 8, 1, 2)$, the original schedule of the backup jobs in the spare processor with non-postponed release time is shown in Figure 5(a). 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 equation (4), $\theta_{11} = 10 - 3 - 0 = 7$. Similarly, θ_{12} can be calculated as 7 as well. So according to equation (5), $\theta_1 = 7$. After that, the release time of all backup jobs of task τ'_1 should be revised according to equation (3), as shown in Figure 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 equation (4), $\theta_{21} = \max\{15 - (8+3) - 0, 7 - (8+0) - 0\} = 4$. Since for this particular example, there is only one backup job in τ'_2 within its hyper period ($LCM_{q\leq 2}(k_q P_q) = 30$), $\theta_2 = 4$. After that the release times of all backup jobs in τ'_2 are postponed by 4 time units according to equation (3). The schedule based on the postponed release times of all jobs within the first hyper period is shown in Figure 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 equation (2), *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 (the proof is provided in the Appendix part):

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 R-pattern.

V. Evaluation

Three different approaches are studied. In the first approach, the task sets are statically partitioned with R-patterns, and the mandatory jobs in the primary and the spare processors are executed concurrently without procrastination. We refer this approach as ($MKSS_{ST}$) and use its results as the reference. The second approach ($MKSS_{DP}$) also determines the mandatory jobs based on the static R-patterns and the mandatory jobs are scheduled with the preference oriented scheme based on dual priority, similar to that used in [8] (but without applying DVS). The third approach ($MKSS_{selective}$) is our selective approach proposed in Section IV based on selective execution of the optional jobs on both the primary and the spare processors. We assume the processor shut-down break even time $T_{be} = 1ms$.

The periodic task set in our experiments consists of five to ten tasks with the periods randomly chosen in the range of $[5, 50]ms$. The m_i and k_i for the (m,k)-deadlines were also randomly generated such that k_i is uniformly distributed between 2 to 20, and $0 < m_i < k_i$. The worst case execution time (WCET) of a task was assumed to be uniformly distributed and the total (m,k)-utilization, *i.e.*, $\sum_i \frac{m_i C_i}{k_i P_i}$, 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 check the energy performance when no fault occurred within the hyper period. The results are shown in Figure 6(a).

From Figure 6(a), one can immediately see that, by adopting dynamic patterns, $MKSS_{selective}$ 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 $MKSS_{selective}$ over $MKSS_{DP}$ can be around 28%. The main reason is that, in this scenario, by executing the optional jobs, $MKSS_{selective}$ can help drop duplicate executions of the mandatory jobs in two processors significantly. Moreover, with adaptive optional job selection/execution strategy, *i.e.*, by only executing optional job of

Demo Version, http://www.verydoc.com and http://www.verypdf.com

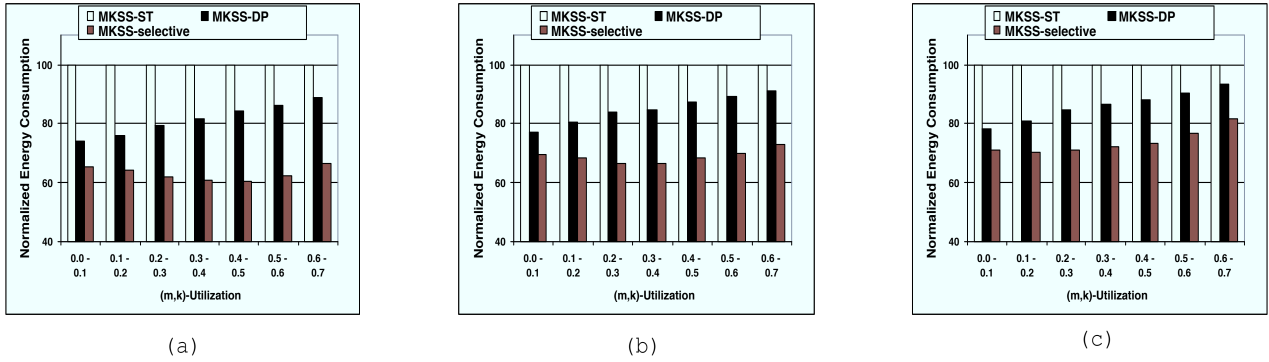

Fig. 6. The energy comparisons for systems under: (a) no fault occurred; (b) permanent fault; (c) permanent and transient faults.

each task with flexibility degree of 1 and letting them be executed in two different processors alternatively, $MKSS_{selective}$ can avoid executing excessive number of the optional jobs. In addition, by letting the backup jobs be delayed with the postponed release times, $MKSS_{selective}$ 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 Figure 6(b).

As seen in Figure 6(b), the energy reduction by our new approaches, *i.e.*, $MKSS_{selective}$ subject to permanent fault is similar to the case when no fault ever occurred. Compared to $MKSS_{DP}$, the energy saving by $MKSS_{selective}$ can be up to 22% 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 in [1] by assuming Poisson distribution with an average fault rate of 10^{-6}. The results were shown in Figure 6(c).

As seen, the energy saving by our new approaches, *i.e.*, $MKSS_{selective}$ in this scenario is similar to that in the previous cases. The maximal energy reduction by $MKSS_{selective}$ over $MKSS_{DP}$ can be up to 16%, thanks to the adaptive executions of the optional jobs under dynamic pattern adjustment.

VI. Conclusion

Energy consumption, QoS, and fault tolerance are among the most critical factors in real-time systems design. In this paper, we presented a novel approach to reduce the energy consumption while assuring (m,k)-deadlines and fault tolerance in standby-spare systems. As shown, the proposed approach outperformed the previous research significantly in energy conservation while ensuring the (m,k)-deadlines and fault tolerance for fixed-priority real time applications.

Acknowledge*

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

References

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

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

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

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

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

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

[7] ——, "Energy-aware standby-sparing for fixed-priority real-time task sets," *Sustainable Computing: Informatics and Systems*, vol. 6, pp. 81 – 93, 2015.

[8] R. Begam, Q. Xia, D. Zhu, and H. Aydin, "Preference-oriented fixed-priority scheduling for periodic real-time tasks," *J. Syst. Archit.*, vol. 69, no. C, pp. 1–14, Sep. 2016.

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

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

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

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

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

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

Appendix

Proof: The correctness of the release postponement interval θ_i for each backup task τ_i' (and its individual backup jobs) is guaranteed by equation (4) and (5) because according to (4) and the definition of J_{ij}-inspecting point in Section IV, the completion time of any backup job will not 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 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. □

Demo Version, http://www.verydoc.com and http://www.verypdf.com