Real-Time Syst (2014) 50:592–619
DOI 10.1007/s11241-014-9210-z

Energy efficient fault-tolerant earliest deadline first scheduling for hard real-time systems

Qiushi Han · Linwei Niu · Gang Quan · Shaolei Ren · Shangping Ren

Published online: 28 September 2014

Abstract Aggressive technology scaling has dramatically increased the power density and degraded the reliability of embedded real-time systems. The goal of our research in this paper is to develop effective scheduling methods that can minimize the energy consumption and, at the same time, tolerate up to K transient faults when executing a hard real-time system scheduled according to the EDF policy. Three scheduling algorithms are presented in this paper. The first algorithm is an extension of a well-known fault oblivious low-power scheduling algorithm. The second algorithm intends to minimize the energy consumption under the fault-free situation while reserving adequate resources for recovery when faults strike. The third algorithm improves upon the first

Q. Han (✉) · G. Quan
Department of Electrical and Computer Engineering, Florida International University,
10555 West Flagler Street, Miami, FL 33174, USA
e-mail: qhan001@fiu.edu

G. Quan
e-mail: gaquan@fiu.edu

L. Niu
Department of Math and Computer Science, West Virginia State University,
5000 Fairlawn Ave, Dunbar, WV 25112, USA
e-mail: lniu@wvstateu.edu

S. Ren
School of Computing and Information Sciences, Florida International University,
10555 West Flagler Street, Miami, FL 33174, USA
e-mail: sren@cs.fiu.edu

S. Ren
Department of Computer Science, Illinois Institute of Technology,
Stuart Building, Room 013C, 10 W. 31st Street,
Chicago, IL 60616, USA
e-mail: ren@iit.edu

two by sharing the reserved resources and thus can achieve better energy efficiency. Simulation results show that the proposed algorithms consistently outperform other related approaches in energy savings.

Keywords EDF · Fault tolerance · Energy consumption · Shared recovery

1 Introduction

As the aggressive scaling in transistor size continues and more and more transistors are integrated into a single die, the power consumption of the IC chips has been increasing exponentially Skotnicki (2005). This in turn poses severe constraints on operations of real-time systems, especially the battery-operated systems due to their limited energy supply. For the past two decades, extensive power management techniques (e.g. Aydin et al. 2004; Yao et al. 1995; Zhang et al. 2003; Mochocki et al. 2004; Quan and Hu 2001; Quan and Niu 2004) have been developed on energy minimization for real-time systems. Among these techniques, dynamic voltage and frequency scaling (DVFS) is one of the most popular and widely deployed schemes. Most modern processors, if not all, are equipped with DVFS capabilities, such as Intel Xeon Intel (2012) and AMD G-series AMD (2014). DVFS dynamically adjusts the supply voltage and working frequency to reduce power consumption at the cost of extended circuit delay. Therefore, for real-time systems with stringent timing constraints, care must be taken to make sure these constraints are satisfied.

In the meantime, as embedded real-time systems grow rapidly in both scale and complexity, reliability is becoming a major concern. First, aggressive technology scaling has significantly decreased the reliability of processors (Skotnicki 2005; Srinivasan et al. 2004) and made electronic devices more susceptible to radiation-induced faults (Lawrence 2007; Langley et al. 2003). Second, high power consumption leads to high operating temperature, which further undermines the reliability of real-time systems. Due to the nature of real-time systems, especially the safety-critical systems such as avionics and industrial controls, catastrophic consequences may occur if system faults are not handled in a timely manner.

Processor faults can be largely classified into two categories: transient and permanent faults (Srinivasan et al. 2003). Transient faults refer to the temporary errors in computation and corruption in data caused by factors such as electromagnetic interference and cosmic ray radiations. On the contrary, permanent faults refer to that the processors are damaged and halt permanently. It may occur due to process and manufacturing defects or wear-out (Srinivasan et al. 2003). In this paper, we focus our research on the transient fault since transient faults occur more frequently than permanent faults (Castillo et al. 1982). Transient faults are usually addressed through *time redundancy* and *backward error recovery* (Zhao et al. 2004). Both techniques require reserving computing resources in case of faults, which may undermine the feasibility of a real-time system. As such, many papers have been published on the feasibility analysis under transient fault conditions for different systems and fault models (e.g. David and Burns 2007; Many and Doose 2011; Aydin Aydin (2007); Pop et al. Pop et al. (2007); Izosimov et al. 2012).

In this paper, we are interested in developing scheduling techniques to minimize the energy consumption and enhance the reliability of a real-time system. This is particularly important in the design of systems, such as surveillance and satellite systems, that demand both energy efficiency and fault tolerance. Recently, the problem to address energy conservation with reliability improvement has drawn considerable attention from many researchers.

When considering the reliability requirement, one approach is to formulate the reliability of a real-time system analytically. For example, Zhao et al. (2004) formulated the reliability of a real-time system as the probability to complete executions of all tasks, with or without fault occurrences. They also proposed a linear and an exponential model to capture the effects of DVFS on transient fault rate and showed that energy management through DVFS could reduce the system reliability. Based on this model, they proposed a recovery scheme to schedule real-time tasks that can reduce energy consumption without degrading the reliability. They further proposed to reserve computing resources that can be shared by different tasks to improve the energy-saving performance Zhao et al. (2011). These algorithms work only for frame-based real-time systems, i.e. tasks with same arrival times and deadlines. Zhao et al. 2012 considered a more general real-time periodic task model. Different tasks may have different periods. For each task, its deadline is equal to its period. Algorithms were proposed to determine the processor speed and resource reservation for each task to achieve the goal of energy minimization under the task-level reliability requirement. The advantage of this approach is that the reliability can be quantified and the impacts of DVFS to reliability can also be taken into consideration. However, to precisely identify the parameters for the reliability model can be challenging, especially when faults usually occur in a burst manner Many and Doose (2011).

Another more intuitive approach is to require that a system can still function properly as long as fault occurrences do not exceed a predefined number. For example, Zhang et al. 2006 introduced a combination of checkpointing and DVFS scheme for tolerating K faults for periodic task sets while minimizing energy consumption. To guarantee the timing constraints, they incorporated the worst case fault recovery time into fixed-priority exact timing analysis to obtain the worst case response time, based on which the energy efficient schedule is determined. Melhem et al. 2004 investigated the same problem for periodic task sets scheduled under EDF on a single processor, assuming $K = 1$. Wei et al. 2012 further extended the approach in Zhang and Chakrabarty citeyearZhang2006TC for the development of combined offline and online DVFS schedules. Since the probability of fault occurrence can be very small, the energy saving performance of the proposed algorithm can be limited. Liu et al. 2010 proposed a heuristic scheduling algorithm that minimizes the energy consumption under the fault-free scenarios and preserves feasibility under the worst case fault occurrences, i.e. up to K faults occur during an operational cycle of the system. This algorithm can only be applied for frame-based real-time task sets. For frame-based real-time task sets, reserved computing resources can be readily shared by different jobs. However, if jobs have different priorities and deadlines, to share the reserved resources becomes much more challenging.

In this paper, we study the problem of minimizing energy consumption and tolerating up to K transient faults when scheduling a set of aperiodic real-time jobs under the

preemptive earliest deadline first (EDF) policy. The solution can be readily extended to a general periodic task set by applying our approaches to the job sets consisting of jobs within the first least common multiple of task periods. Three algorithms are proposed in the paper. The first two algorithms are developed based on the well-known algorithm presented in Yao et al. 1995 (so called LPEDF). The third algorithm improves upon the first two by sharing the reserved computing resources and thus can achieve better energy saving performance. Simulation results show that the proposed algorithms can significantly reduce the energy consumption when compared with related approaches.

The rest of the paper is organized as follows. In Sect. 2, we introduce system models and formally formulate our research problem. Our three algorithms are presented in Sects. 3 and 4. Section 5 extends our algorithms to deal with several practical issues. Section 6 discusses our simulation results. Finally, we conclude our paper in Sect. 7.

2 Preliminaries

In this section, we first introduce the system models and related notations. We then formulate our problem formally.

2.1 Real-time application model

We model a real-time system as a job set $\mathcal{J} = \{J_1, J_2, ..., J_n\}$, where J_i denotes the i_{th} job in a job set and is characterized by a tuple (a_i, c_i, d_i). The definition of theses parameters is given in the following:

- a_i: The time when J_i is ready for execution, referred to as arrival time;
- c_i: The worst case execution time of J_i under s_{max}, where s_{max} is the maximum speed that the processor supports;
- d_i: The absolute deadline of J_i.

This model is rather general and can be readily extended to other real-time models such as the general periodic task model. All jobs are considered to be independent and scheduled under preemptive EDF policy on a single processor.

2.2 Power and energy model

For ease of our presentation, we assume the speed/frequency (two terms are used interchangeably throughout the paper) of a processor can be changed continuously in $[s_{min}, s_{max}]$ with $0 \leq s_{min} \leq s_{max} = 1$. Later in this paper, we extend our algorithms to processors supporting only a set of discrete levels of processor speed. A job is assumed to execute with only one speed. Therefore, when J_i is executed under speed s_i, the execution time of J_i becomes $\frac{c_i}{s_i}$. A speed schedule for an entire job set is denoted as $S = \{s_1, s_2, s_3, ..., s_n\}$ where s_i is the speed for J_i.

Our system-level power model is similar to that in Zhao et al. (2011) by distinguishing the frequency-independent and frequency-dependent power components. Specifically, the overall power consumption (P) can be formulated as

$$P = P_{ind} + P_{dep} = P_{ind} + C_{ef} s^{\alpha} \tag{1}$$

where P_{ind} is the frequency-independent power, including the power consumed by off-chip devices such as main memory and external devices and constant leakage power. C_{ef} is the effective switching capacitance. α is a constant usually no smaller than 2. P_{dep} is the frequency-dependent active power, including the CPU power, and any power that depends on the processing speed s. Hence, the energy consumption of a job J_i running at the speed s_i can be expressed as:

$$E_i(s_i) = \left(P_{ind} + C_{ef} s_i^{\alpha}\right) \cdot \frac{c_i}{s_i} \tag{2}$$

As $E_i(s_i)$ is a convex function, the minimum system energy is achieved when s_i is as small as possible, provided it is larger than so-called *critical speed* (s_c) Zhao et al. (2004). In this study, we assume that $s_{min} \geq s_c$.

2.3 Fault model

We assume that the system is subject to a maximum of K transient faults (e.g., bit flips in architectural registers or timing errors in CMOS circuit). Faults usually are detected at the end of each job J_i's execution using *acceptance* or *sanity tests* Pradhan (1996) and the timing and energy overhead for detection are denoted as TO_i and EO_i, respectively. Furthermore, we assume that the overheads of fault detections are not subject to frequency variations. There is no assumption regarding the occurrence pattern of faults, i.e. faults can occur anywhere at any time during an operational cycle of the system, multiple faults may hit a single job. A fault is tolerated by re-executing the affected job. Therefore, the maximum recovery overhead for job J_i executing at s_{max} under a single failure, denoted as R_i, is c_i, or $R_i = c_i$. When a fault happens during the execution of J_i, a recovery job that of the same deadline d_i is released. The recovery jobs are subject to preemption as well.

2.4 Problem formulation

We formulate our problem formally as follows:

Problem 1 Given a real-time job set $\mathcal{J}$ scheduled under EDF on a single processor, find a speed schedule S for all the jobs in $\mathcal{J}$ (including the recoveries) such that the processor energy consumption is minimized without any deadline miss when no more than K faults occur.

3 Fault tolerant speed schedule

In this section, we introduce an approach to the development of a fault tolerant DVFS schedule for a hard real-time job set to reduce the energy consumption. The algorithm is developed based on LPEDF presented in Yao et al. (1995). To ease the presentation of our approach, we first introduce several definitions and then reiterate briefly the general idea of LPEDF.

Definition 1 Given a real-time job set $\mathcal{J}$,

- $\mathcal{J}(I)$ denotes the set of jobs contained in the interval $I = [t_s, t_f]$, i.e. $\mathcal{J}(I) = \{J_i | t_s \leq a_i < d_i \leq t_f\}$;
- The workload $W(I)$ of an interval $I = [t_s, t_f]$ is the accumulated execution time of jobs completely contained in the interval, i.e $W(I) = \sum_{J_i \in \mathcal{J}(I)} c_i$;
- The intensity of interval I is defined as

$$s(I) = \frac{W(I)}{L(I)}, \tag{3}$$

  where $L(I)$ is the length of interval I, i.e. $L(I) = t_f - t_s$;
- The interval $I = [t_s, t_f]$ is called a critical interva if it has the highest intensity and t_s and t_f are the arrival time and the deadline of some job(s), correspondingly.
- The fault-related overhead of an interval I is denoted as $W_{ft}(I) = W_r(I) + W_{TO}(I)$, where $W_r(I)$ represents the reserved workload to be used for recovery in the worst case, i.e. $W_r(I) = K \times (R_x + TO_x)$ and x denotes the index of the job with the longest recovery time in $\mathcal{J}(I)$, i.e. $J_x = \{J_i | \max(R_i + TO_i), J_i \in J(I)\}$ and $W_{TO}(I)$ denotes the overhead imposed by fault detections from regular jobs, i.e. $W_{TO}(I) = \sum_{J_i \in \mathcal{J}(I)} TO_i$.

Given a real-time job set $\mathcal{J}$, LPEDF can be employed to minimize the energy consumption (assuming $s_{min} \geq s_c$) as follows Yao et al. (1995):

1. Step 1: Identify a critical interval $I = [t_s, t_f]$ using Eq. (3);
2. Step 2: Remove the critical interval and all jobs contained in the interval, set the speeds of all jobs in $\mathcal{J}(I)$ to $s(I)$ and modify the arrival times and deadlines of other jobs accordingly. Specifically, let $\mathcal{J} \leftarrow \mathcal{J} - \mathcal{J}(I)$; change deadline d_i to t_s if $d_i \in [t_s, t_f]$, or to $d_i - (t_f - t_s)$ if $d_i \geq t_f$; set a_i to t_s if $a_i \in [t_s, t_f]$, or to $a_i - (t_f - t_s)$ if $a_i \geq t_f$.
3. Step 3: Repeat step (1)–(2) until $\mathcal{J}$ is empty.

To make the above LPEDF fault-tolerant, one intuitive approach (we call this approach as MLPEDF) is to take the fault recovery into consideration and increase the workload of an interval when calculating its intensity, that is, to replace $s(I)$ with $s_m(I)$, as defined in Eq. (4),

$$s_m(I) = \frac{W(I) + K \times R_x}{L(I) - W_{TO}(I) - K \times TO_x}, \tag{4}$$

where x is the index of the job with longest recovery in $\mathcal{J}(I)$ and $W_{TO}(I)$ denotes the total fault-detection overheads for regular jobs as defined in Definition 1.

We summarize the feasibility condition of an arbitrary EDF-scheduled job set on a single processor that is subject to a maximum number of K transient faults in the following.

Theorem 1 Aydin (2007) *Given a real-time job set $\mathcal{J}$ with K faults to be tolerated and $s_{max} = 1$, if for each interval I, we have*

Table 1 A real-time system with three jobs

	a_i	c_i	d_i
J_1	0	1	9
J_2	7	3	15
J_3	13	1	20

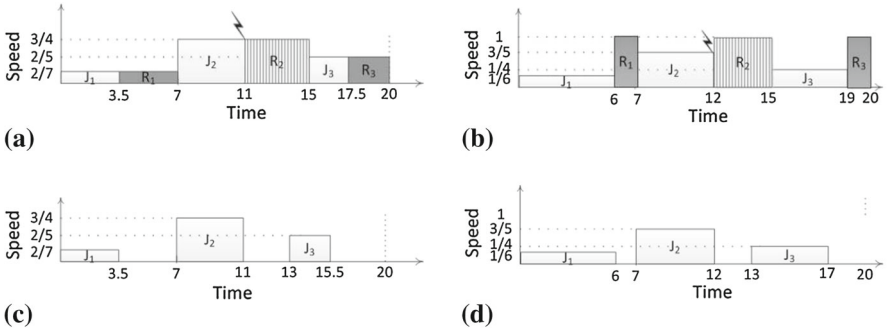


Fig. 1 MLPEDF versus EMLPEDF. K is set to 1, a *dark grey rectangle* represents a reserved recovery block and a *shaded rectangle* indicates that a recovery block becomes active, i.e. a fault has been encountered. **a** and **b** show the schedules when the fault affects the job with the longest execution time, i.e. J_2 under MLPEDF and EMLPEDF, respectively. The reserved recovery blocks are not shown in the fault-free schedules. **a** Fault recovery schedule under MLPEDF. **b** Fault recovery schedule under EMLPEDF. **c** Fault free schedule under MLPEDF. **d** Fault free schedule under EMLPEDF

$$\frac{W(I) + W_{ft}(I)}{L(I)} \leq 1, \tag{5}$$

then the job set $\mathcal{J}$ is feasible.

Note that, when a fault occurs, MPLEDF executes the recover copy of a job using a scaled processor speed. This helps to reduce the total energy consumption for both the original jobs and their recovery copies. However, this may not be energy efficient in a practical scenario when the possibility of fault occurrence is low.

An alternative approach (we call this approach as EMLPEDF), is to run the recovery backups using the maximum possible processor speed. The intensity calculation of interval I can be modified correspondingly, as Eq. (6).

$$s_e(I) = \frac{W(I)}{L(I) - W_{ft}(I)} \tag{6}$$

It can be easily verified that $s_e(I) \leq s_m(I)$ for a given interval I if $W(I)+W_{ft}(I) \leq L(I)$, which always holds for a feasible schedule. The advantage of this approach is that it requires the least amount of resource reservation to guarantee the timely recovery, and thus can reduce the energy consumption under the fault-free scenario. This can be further illustrated using the following example.

Consider the simple real-time job set, shown in Table 1 and Fig. 1. Let $\alpha = 2$, $P_{ind} = 0.02$ and $C_{ef} = 1$. For simplicity, the timing and energy overhead are considered

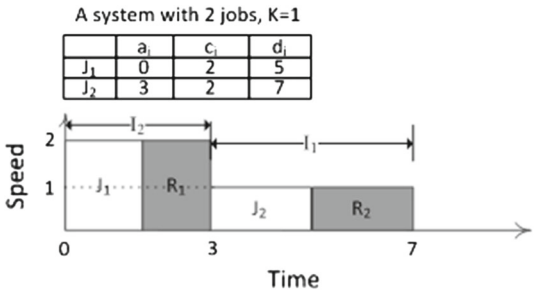


Fig. 2 Monotonicity violation example

negligible. We can calculate that the fault recovery schedule by MLPEDF (Fig. 1a) consumes less energy than that by EMLPEDF (Fig. 1b), i.e. 5.41 vs. 5.56. However, the fault-free schedule by MLPEDF (Fig. 1c) consumes much more energy than that by EMLPEDF (Fig. 1d), i.e. 3.08 versus 2.48 (20 % more). Since the fault rate is usually very low in practice, EMLPEDF can have a much better energy saving performance than MLPEDF.

To ensure the deadlines, when removing a critical interval and updating the arrivals or deadlines of remaining jobs in each iteration (similar to each round of Step 1 and Step 2 in LPEDF), we assume that all K faults will affect the longest job in the critical interval under the worst case. This assumption is rather pessimistic because each critical interval demands computing resources reserved for tolerating K faults, which may potentially cause a feasible job set infeasible. We use an example to illustrate this problem.

Consider a system with two jobs specified in Fig. 2 and at most one fault to be tolerated. For ease of presentation, we set the overheads of fault detections to 0. According to EMLPEDF, the first critical interval is interval [3,7] with intensity 1 based on Eq. (6). After the removal of interval [3,7] along with job J_2, d_1 is updated as 3 and the second critical interval is [0,3] with intensity 2. We have the schedule drawn in Fig. 2, where I_1 and I_2 denote the first and second critical interval, respectively. We can see that $s_e(I_2)$ is larger than $s_e(I_1)$(we refer to this situation as the monotonicity violation). Moreover, $s_e(I_2)$ exceeds the highest speed available in the system ($s_{max} = 1$), so the required speed is unachievable. However, it is not hard to see that the job set is in fact feasible under constant speed 1. From the above discussion, it is clear that the energy minimization problem with fault tolerance requirement cannot be solved by simply modifying the LPEDF solution. Provisions are required during the scheduling process to ensure that the resulting schedule is valid.

To handle monotonicity violations, we observed that any critical interval that violates monotonicity must be adjacent to the critical interval found in the previous iteration. Specifically, we have the following lemma.

Lemma 1 *Let I_i and I_{i-1} be two critical intervals identified by EMLPEDF from i_{th} and $(i-1)_{th}$ iteration*[1]*, respectively. If $s_e(I_i) > s_e(I_{i-1})$, I_i and I_{i-1} are adjacent.*

[1] Each iteration of EMLPEDF refers to one round of the Step 1–2 in LPEDF except the intensity function is defined in Eq. (6).

Proof When removing interval I_{i-1}, the workload distribution is not changed in the intervals that have no overlap with I_{i-1}. Only the intervals overlapping I_{i-1} are shortened by Δ, $0 < \Delta \leq L(I_{i-1})$; therefore, they may experience an increase in intensity in the next iteration. □

As implied in the proof of Lemma 1, a monotonicity violation occurs when the removed critical interval contains slacks that need to be reserved as recoveries for jobs in its overlapping intervals. Therefore, the execution space for these jobs are shortened due to its removal. To eliminate such monotonicity violations, we can incorporate these jobs into the previously found critical interval. We formulate this conclusion in Lemma 2.

Lemma 2 *Let I_i and I_{i-1} be two critical intervals identified by EMLPEDF from i_{th} and $(i-1)_{th}$ iteration, respectively. If $s_e(I_i) > s_e(I_{i-1})$, the minimum constant speed to maintain feasibility of jobs contained in I_i and I_{i-1} is $s_e(I_{i-1})$.*

Proof Before removing the critical interval I_{i-1} ($i > 1$), all the remaining jobs in $\mathcal{J}$(jobs left after first $i-2$ iterations) are feasible under the constant speed $s_e(I_{i-1})$. Therefore, the combined jobs in I_i and I_{i-1} are definitely feasible under this speed. □

Lemma 1 and Lemma 2 help us to keep track of the monotonicity violation and remove it whenever it occurs.

Up to now, we can formulate our EMLPEDF algorithm in Algorithm 1. Line 4 identifies the current critical interval and its speed. Lines 5–8 check if the current desired speed is less than the minimal available speed, and terminate the iteration if so. Lines 9–12 remove monotonicity violation whenever it occurs. Line 14 backs up the timing information of jobs in case a rollback operation is needed. Lines 15–17 remove the critical interval and update the job set. The complexity of EMLPEDF mainly comes from calculations of critical intervals (line 4), i.e. $O(n^2)$ with a straightforward implementation. The overall complexity of EMLPEDF is same as LPEDF, i.e. $O(n^3)$.

We have the following theorem regarding the lowest constant speed that guarantees the feasibility of a job set.

Theorem 2 *Let $s_{e1}, s_{e2}, s_{e3}, \ldots$ be the intensities for the critical intervals from iteration 1, 2, 3... in EMLPEDF. s_{e1} is the lowest constant speed that can be employed throughout the entire job set without causing any deadline miss as long as no more than K faults happen.*

Proof This theorem can be proved directly in light of Theorem 1. During the first iteration of EMLPEDF, we have $\frac{W(I)}{s_{e1}} \leq \frac{W(I)}{s_e(I)}$ for each interval I, since $s_{e1} \geq s_e(I)$ considering the definition of critical interval. Take Eq. (6) into the right-hand side of the above inequality and add $W_{ft}(I)$ to both sides. We have $\frac{W(I)}{s_{e1}} + W_{ft}(I) \leq L(I)$. Therefore, the job set is feasible under constant speed s_{e1}.

Moreover, assume s_{e1} is the resulting intensity from interval I_1, i.e. $s_{e1} = \frac{W(I_1)}{L(I_1)-W_{ft}(I_1)}$ and s^* is the lowest constant speed that maintains the feasibility of the job set and $s^* < s_{e1}$. We have the scaled workload in I_1 as $\frac{W(I_1)}{s^*} + W_{ft}(I_1) > \frac{W(I_1)}{s_{e1}} + W_{ft}(I_1) = L(I_1)$, which violates the feasibility condition in Theorem 1. □

In addition, by applying Algorithm 1, we have the following theorem regarding the characteristics of critical interval speeds.

Algorithm 1 EMLPEDF algorithm

Require:
1) Job set : $\mathcal{J} = \{J_1, J_2, ...J_n\}$;
2) Number of faults: K
3) minimum frequency available: s_{min}
1: $s_i = s_{max}$, for $i = 1, 2, ..., n$;
2: $p = 1$; {the critical interval index}
3: **while** $\mathcal{J} \neq \emptyset$ **do**
4: Identify the next critical interval $I_p^* = [t_s, t_f]$ and its intensity $s_{e,p}$ based on Eq. (6);
{$s_{e,p}$: the intensity of p_{th} critical interval}
5: **if** $s_{e,p} < s_{min}$ **then**
6: $s_i = s_{min}, \forall J_i \in \mathcal{J}$;
7: break;
8: **end if**
9: **if** $s_{e,p} > s_{e,p-1}$ AND $p > 1$ **then**
10: Restore the timing information from the previous iteration;
11: Merge the interval I_p^* with I_{p-1}^*;
12: $p - -$;{Roll back the critical interval index}
13: **end if**
14: Back up the timing information;
15: $s_i = s_{e,p}, \forall J_i \in \mathcal{J}(I_p)$;
16: remove all jobs in I_p from $\mathcal{J}$;
17: update timing information of remaining jobs according the step 2 in LPEDF Yao et al. (1995)
18: $p + +$;
19: **end while**
20: **return** $\{s_1, s_2, ..., \}$

Theorem 3 *Let $s_{e1}, s_{e2}, ...s_{em}$ be the intensities for the critical intervals from iteration 1, 2, ...m in EMLPEDF. We have $s_{e1} \geq s_{e2} ... \geq s_{em}$.*

Proof Because all monotonicity violations are eliminated in Algorithm 1, the non-increasing relationship between subsequent critical intervals can be easily determined. □

More importantly, if EMLPEDF can be successfully applied for a job set, then the feasibility of the result DVFS schedule is guaranteed. This is summarized in the following theorem.

Theorem 4 *EMLPEDF can guarantee that all jobs can meet their deadlines as long as the following two constraints are satisfied : (1) no more than K faults occur; (2) $\forall i \in [1, m]$, where m is the total number of iterations, we have $s_{ei} \leq 1$.*

Proof In EMLPEDF, a critical interval I_i is exclusively reserved for executing jobs and their recovery copies in the interval. For any higher priority job (e.g. J_h) with possible execution overlapping with I_i, it is forced to finish before the I_i in EMLPEDF. Similarly, for any lower priority job (e.g. J_l) with possible execution overlapping with I_i, the interval I_i is excluded for its execution by adjusting its arrival time and deadline in EMLPEDF. Therefore, to prove the theorem, we only need to prove that if we set the processor speed to be s_{ei}, i.e. the intensity of I_i, throughout I_i, then the schedulability of all jobs in I_i is guaranteed in the worst case (i.e. against K faults), as long as $s_{ei} \leq 1$.

We prove this by contradiction. Let $J_c = (r_c, c_c, d_c) \in \mathcal{J}(I_i)$ miss its deadline when processor speed is set to s_{ei}. Then we must be able to find a time $t \leq r_c$, such that for

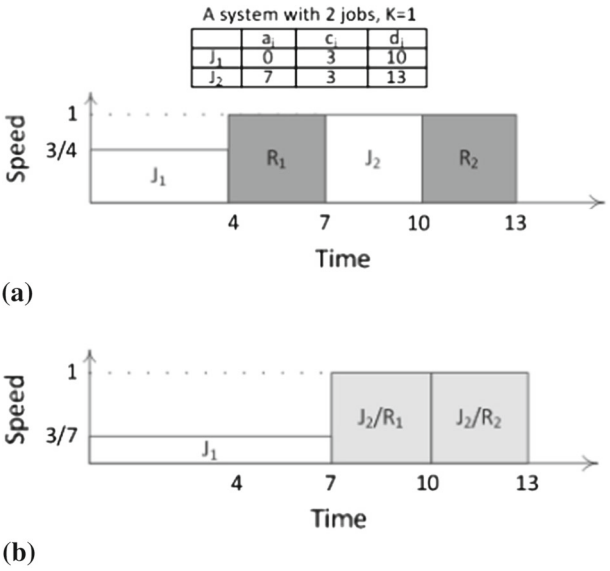


Fig. 3 EMLPEDF versus LPSSR. **a** Speed schedule by EMLPEDF. **b** Speed schedule by LPSSR

interval $I' = [t, d_c]$, we have $\frac{W(I')}{s_{ei}} + W_{ft}(I') > L(I')$. Since $s' = \frac{W(I')}{L(I') - W_{ft}(I')} > s_{ei}$ and $I' \subseteq I_i$. This violates the assumption that I_i is a critical interval.

Since all jobs are associated with a critical interval in EMLPEDF and all jobs within a critical interval are schedulable when the corresponding speed is applied, we prove the theorem. □

While EMLPEDF can guarantee the feasibility of a real-time job set under maximum K faults, and can also achieve better energy saving performance than MLPEDF, each critical interval needs to reserve computing resource separately for timely recovery when faults happen. It is desirable that different critical intervals can share the reserved resources and conceivably the energy saving performance can be further improved. We develop a new algorithm for this purpose, which is introduced in the coming section.

4 Fault tolerant speed schedule with shared recovery slacks

This section presents an improved approach to the development of energy efficient fault tolerant schedule for a given job set $\mathcal{J}$. We call this algorithm LPSSR. Specifically, LPSSR improves upon EMLPEDF by allowing different critical intervals to share reserved computing resources. We also execute recovery under s_{max} in LPSSR and focus on determining the speed schedule S for regular jobs. Before we introduce the algorithm in details, we first use an example to motivate our research.

Consider a simple job set with two jobs specified in Fig. 3. Note that, we set the overheads of fault detection to 0 for easy presentation. The speed schedule by EMLPEDF is shown in Fig. 3a. Note that in Fig. 3a, interval R_1 (i.e. interval [4, 7])

and interval R_2 (i.e. interval [10, 13]) are the recovery blocks used for fault recovery. However, since $K = 1$, at most one of the recovery blocks can be used. If the fault occurs during J_1's execution, R_1 will be used for recovery. In that case, R_2 will never be used since no fault will happen during J_2's execution. Same problem occurs if the fault affects J_2's execution.

A better fault tolerant schedule is shown in Fig. 3b. Note that, when the fault affects J_1, the interval [7, 10] can serve as the reserved block to run the backup of J_1, and J_2 can be executed at interval [10, 13]. If the fault affects J_2, since there is no fault during J_1's execution, J_2 can be executed at interval [7, 10], and later recovered at interval [10, 13] if necessary. For either case, the system is always feasible. By sharing the recovering slacks, the speed of J_1 is reduced to $3/7$. Using the same system parameters as in the previous example, the energy consumption of the new schedule is more than 30 % lower than that by EMLPEDF. The example clearly shows that significant energy savings can be obtained without compromising the system feasibility if the reserved computing resource can be shared. The problem is how to judiciously share the reserved resource to maximize the energy saving performance. In what follows, we develop an approach to explore the shared slacks to improve the energy efficiency.

When removing a critical interval in EMLPEDF, its reserved slacks can only be shared by jobs that have potential execution overlaps with it. To ease our presentation, we classify these jobs into the following categories as defined below.

Definition 2 For a given interval $I = [t_s, t_f]$ a job J_i is referred to as deadline overlapping with I if $a_i \notin [t_s, t_f]$ and $d_i \in [t_s, t_f]$, and arrival overlapping with I if $a_i \in [t_s, t_f]$ and $d_i \notin [t_s, t_f]$, and fully overlapping with I if $I \subseteq [a_i, d_i]$.

Specifically, for interval I, we denote all deadline overlapping jobs, arrival overlapping jobs, and fully overlapping jobs as $\mathcal{J}_I^{do}$ and $\mathcal{J}_I^{ao}$, and $\mathcal{J}_I^{fo}$, respectively.

In EMLPEDF, when a critical interval is identified, it is removed with all jobs inside it to make sure that the interval is exclusively used for running jobs and their backups that are completely located within the interval. Also, the arrival times and deadlines of the others are updated to the boundary of the interval such that their executions will never interfere with jobs in the critical interval. In LPSSR, we allow a job to share the reserved slacks in the critical interval by "extending" its deadline or arrival time "into" the critical interval. We discuss each category of jobs separately as follows.

Let $I^* = [t_s, t_f]$ be a critical interval with length $L(I^*)$, and intensity $s_e(I^*)$, which is calculated the same way (i.e. Eq. (6)) as that in EMLPEDF. Let $R_{max}(I^*) = W_r(I^*)$, $R_{min}(I^*) = K \times \min\{R_j + TO_j | J_j \in \mathcal{J}(I^*)\}$ be the upper and lower bound of the reserved slacks. Also, let J_i be a job with execution interval (i.e. $[a_i, d_i]$) partially or fully overlapped with I^*, the overlap length is represented as $L(I_i^{op})$. Additionally, the maximum amount of reserved slack shared by a job J_i is denoted by $RS(J_i)$. Consider the following three cases:

– $J_i \in \mathcal{J}_{I^*}^{do}$: To share the reserved slack, the deadline of J_i will be *extended into* interval I^*. interval depends on how much the reserved slacks can be shared by J_i without compromising the fault tolerant feasibility within the critical interval. Note that, $RS(J_i)$ cannot exceed either $R_{min}(I^*)$ or $K \times R_i$. This is because if we know that the execution of jobs contained in I^* is fault-free, it is safe to delay

the starting of critical interval by $R_{min}(I^*)$ and leave the space to J_i. $R_{min}(I^*)$ is a lower bound of delay a fault-free critical interval can tolerate without deadline misses. If the shared slacks are used by J_i or its recovery during run-time, which means some faults (we assume it to be K_1,$0 < K_1 \leq K$) have hit the jobs before I^*. As a result, $\frac{K_1}{K} \times R_{max}(I^*)$ slacks are reclaimed in I^*. Since the reserved slacks are shared among jobs contained in the critical interval, it virtually means each job $J_j \in J(I^*)$ has reclaimed its own reserved slacks in the amount of $\frac{K_1}{K} \times R_j$ which counters the delay caused by pushing the execution of J_i into the critical interval and the remaining reserved slacks are still enough to handle the rest of upcoming faults. In addition, $RS(J_i)$ can not exceed the total overlap length $L(I_i^{op})$. In conclusion, Specifically, instead of t_s, d_i is set to $t_s + RS(J_i)$ where $RS(J_i) = \min(K \times (R_i + TO_i), K \times R_{min}(I^*), L(I_i^{op}))$.

- $J_i \in \mathcal{J}_{I^*}^{ao}$: In this case, the new arrival time of J_i will be *extended into* interval I^* in order to share the reserved slack. reserved slack shared by J_i cannot exceed either $R_{min}(I^*)$ or $K \times R_i$. Therefore $RS(J_i) = \min(R_{min}(I^*), K \times R_i, L(I_i^{op}))$. To share the slacks, after removing the critical interval I^* (only subinterval $[t_s, t_f - RS(J_i)]$ is effectively removed, where $RS(J_i) = \min(K \times (R_i + TO_i), K \times R_{min}(I^*), L(I_i^{op})))$, we set J_i's deadline as $d_i = d_i - L(I^*) + RS(J_i)$, and update a_i to t_s.
- $J_i \in \mathcal{J}_{I^*}^{fo}$: In this case, all the reserved slacks in I^* can be potentially used by J_i. To share the slack, after removing the critical interval I^*, we set J_i's deadline as $d_i = d_i - L(I^*) + RS(J_i)$, where $RS(J_i) = \min(R_{max}(I^*), K \times (R_i + TO_i))$.

Accordingly, we formulate a new algorithm (i.e. LPSSR), as shown in Algorithm 2. Without loss of generality, we ignore the overheads of fault detections. The work flow of Algorithm 2 is similar to EMLPEDF, i.e. iteratively identifying critical intervals, removing the critical interval and the jobs inside the critical interval, and then updating the timing parameters for the rest of the jobs, until the job queue becomes empty. Different from EMLPEDF, we apply our sharing technique when updating the timing parameters and eliminate monotonicity violation whenever it occurs. In Algorithm 2, line 4 identifies p_{th} critical interval for the current real-time job set. Lines 5–8 are simply the application of Theorem 2. Lines 9–13 roll back to the previous iteration and merge the current critical interval with the previous one once monotonicity violation is found. Lines 16–29 backup and update the timing parameters of each remaining jobs according to the sharing technique discussed above. At last, lines 30–33 remove all jobs inside the critical interval.

The main computation complexity per iteration comes from identifying the critical interval, which is $O(n^2)$, where n is the number of jobs. The outer loop can at most repeat n times. Therefore, the complexity of Algorithm 2 is $O(n^3)$. In what follows, we first use an example to illustrate the procedures of LPSSR. We then prove that the algorithm can guarantee the schedulability of all jobs under K faults.

Consider a system with 5 jobs whose timing information is given in Fig. 4a. We assume that $K = 1$. We use $\uparrow$ and $\downarrow$ to denote a job's arrival time and deadline, respectively. The fault-detection overheads are considered negligible in this example. For each step, the critical interval is identified with intensity function in Eq. (6) and is shown as $| \leftrightarrow |$. For the first iteration in Fig. 4a, the critical interval is identified as

Algorithm 2 LPSSR algorithm

Require:
1) Job set : $\mathcal{J} = \{J_1, J_2, ...J_n\}$;
2) Number of faults: K

$s_i = s_{max}$, for $i = 1, 2, ..., n$;
$p = 1$;{critical interval index}
while $\mathcal{J} \neq \emptyset$ **do**
  Identify the critical interval $I_p = [t_s, t_f]$ and its intensity $s_{e,p}$ based on Eq. (6);
  {$s_{e,p}$: the intensity of p_{th} critical interval}
  if $s_{e,p} < s_{min}$ **then**
    $s_i = s_{min}, \forall i \in \mathcal{J}$;
    break;
  end if
  if $s_{e,p} > s_{e,p-1}$ AND $p > 1$ **then**
    Restore the timing information from the previous iteration;
    Merge the interval I_p with I_{p-1};
    $p - -$;{Roll back the critical interval index}
  end if
  $L(I_p) = t_f - t_s$;
  for all $J_i \in \mathcal{J}$ **do**
    Backup timing information of J_i;
    $RS(J_i) = \min(K \times (R_i + TO_i), K \times R_{min}(I_p), L(I_i^{op}))$;//$R_{min}(I_p)$ is the minimum recovery time for jobs in $\mathcal{J}$
    if $J_i \in J_{I_p}^{do}$ **then**
      $d_i \leftarrow \min\{d_i, t_s + RS(J_i)\}$;
    else if $J_i \in J_{I_p}^{ao}$ **then**
      $d_i \leftarrow d_i - (L(I_p) - RS(J_i))$
      $a_i \leftarrow t_s$;
    else if $J_i \in J_{I_p}^{fo}$ **then**
      $RS(J_i) = \min(K \times (R_i + TO_i), K \times R_{max}(I_p))$;
      $d_i \leftarrow d_i - (L(I_p) - RS(J_i))$;
    else
      $a_i \leftarrow a_i - L(I_p)$;
      $d_i \leftarrow d_i - L(I_p)$;
    end if
    for all $J_q | [a_q, d_q] \subseteq I_p$ **do**
      $s_q = s_{e,p}$;
      $\mathcal{J} \leftarrow \mathcal{J} - \mathcal{J}(I_p)$;
    end for
  end for
  $p + +$;
end while
return $\{s_1, s_2, ..., \}$

[5,10] with intensity $s_e([5, 10]) = \frac{c_1+c_2}{10-5-R_2} = 1$. When we remove interval [5,10], J_1 and J_2 are removed and speed 1 is assigned to both jobs, and then we need to update the timing information of the remaining jobs.

Note that J_3 is a *fully overlapping* job with respect to the critical interval, and all the slacks that reserved in interval [5,10] can be used by J_3. Therefore, $RS(J_3) = \min(R_{max}[5, 10], R_3) = \min(2, 3) = 2$. Consequently, we have $d_3 = d_3 - L([0, 5]) + RS(J_3) = 17$. For J_4, it is *deadline overlapping* with the critical interval and thus $RS(J_4) = \min(R_4, R_{min}([5, 10]), L_i^{op}) = \min(2, 1, 2) = 1$. As a result, d_4 is set to

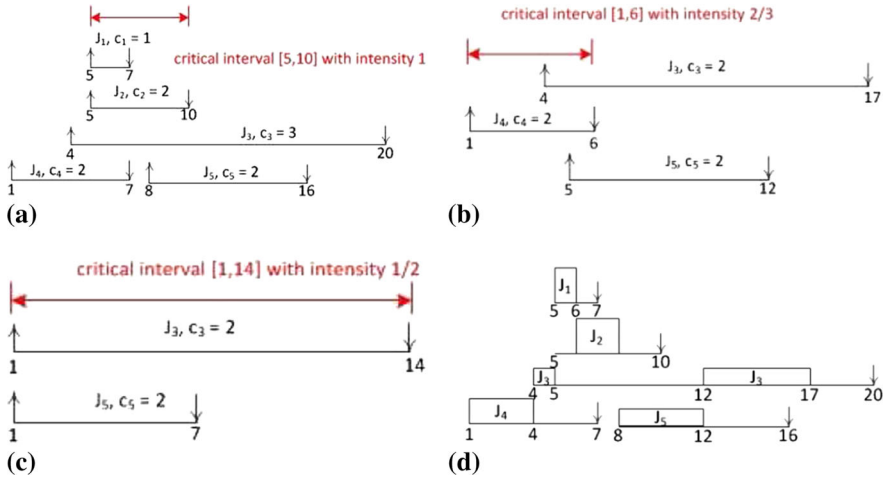


Fig. 4 An example of LPSSR. **a** First iteration of LPSSR. **b** Second iteration of LPSSR. **c** Third iteration of LPSSR. **d** Fault-free schedule under the speeds from LPSSR

6, i.e. the boundary of the critical interval (5) plus the slacks that can be shared by J_4. For J_5, which is a *arrival overlapping* job with respect to the critical interval, the slacks that be shared by J_5 is $RS(J_5) = \min(R_5, R_{min}([5, 10]), L_i^{op}) = \min(2, 1, 2) = 1$ and its arrival and deadline are set to 5 and 12, respectively. The resulting job set is illustrated in Fig. 4b.

Based on the new job set, we identify the critical interval as [0,6] with intensity 2/3. After the assign speed 2/3 to J_4 and remove the critical interval and repeat the same procedures as discussed in the previous iteration, we have a consequent job set as shown in Fig. 4c. Finally, the last critical interval is [1,14] with intensity 1/2 and J_3 and J_5 are removed after being allocated a speed 1/2. The LPSSR algorithm terminates and we have the resulting speed schedule $S = \{1, 1, 1/2, 2/3, 1/2\}$. The final schedule is shown in Fig. 4d, and it can be verified that no matter when the failure occurs, there is no deadline miss with this schedule.

Moreover, the feasibility of the schedule output from Algorithm 2 is guaranteed, which is formulated in Theorem 5.

Theorem 5 *Given a real-time job set $\mathcal{J}$ and a constant K, all the jobs in $\mathcal{J}$ can meet their deadlines if they are executed based on the processor speeds determined by Algorithm 2 and no more than K faults occur.*

Proof The proof of Theorem 5 is similar to that of Theorem 4. Let $I^* = [t_s, t_f]$ be the critical interval and $s_e(I^*)$ be its speed. We consider the three types of jobs separately.

Case 1: let $J_i \in \mathcal{J}_{I^*}^{do}$ and d_i' denote the deadline after the removal of the critical interval I^*, i.e, $d_i' = t_s + RS(J_i)$. If J_i and its recovery workload finishes at t_s or earlier, it has no impact to the execution for jobs in $\mathcal{J}(I^*)$. Hence all jobs in $\mathcal{J}(I^*)$ are schedulable under K faults in the worst case. If J_i and its recovery workload finishes at d_i', this means that all K faults must occur before d_i'. Otherwise, one more fault occurs at d_i' will cause J_i to miss deadline. As a result, there will be no faults occurring in

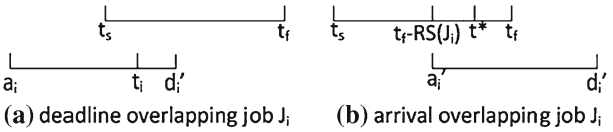


Fig. 5 **a** d_i' is deadline to be assigned after the removal of critical interval, which is $t_s + RS(J_i)$, t_i is the finishing time of J_i or its recoveries. **b** t^* is the completion time of all the jobs and recoveries in $\mathcal{J}(I^*)$, a_i' is extended into I^* by $RS(J_i)$

interval I^*. Since $d_i' - t_s = RS(J_i) \leq K \times R_{min}(I^*)$, this implies that the slack time occupied by J_i is smaller than the minimum amount of reserved slack in interval I^* that can be exploited by every job to execute the recovery workload. Therefore, all jobs in $\mathcal{J}(I^*)$ must be schedulable. The question now becomes what if J_i and its recoveries finishes at t_i, where $t_s < t_i < t_s + RS(J_i)$, refer to Fig. 5a.

We consider the following two cases.

- Case 1-a: $R_i + TO_i >= R_{min}(I^*)$.
  Then there are at most K' faults, where $K' = \lfloor (d_i' - t_i)/(R_i + TO_i) \rfloor$ left after $t > t_i$. Otherwise, if more than K' faults occurring at (or after) t_i will cause J_i to miss its deadline. In other words, there must be $K - K'$ faults occurred before t_i. Note that J_i consumes a slack of $t_i - t_s$ from I^*. In the meantime, each job at least has an additional slack of $(K - K') \times R_{min}(I^*)$ to spare. Since $K \times R_{min}(I^*) \geq d_i' - t_s = RS(J_i)$, we have

$$\begin{aligned}(K - K') \times R_{min}(I^*) &\geq (K - \lfloor (d_i' - t_i)/(R_i + TO_i) \rfloor) \times R_{min}(I^*) \\ &\geq (K - (d_i' - t_i)/(R_i + TO_i)) R_{min}(I^*) \\ &\geq d_i' - t_s - (d_i' - t_i)/R_{min}(I^*) \times R_{min}(I^*) \\ &= t_i - t_s.\end{aligned}$$

  Therefore, all jobs in $\mathcal{J}(I^*)$ can be schedulable.
- Case 1-b: $R_i + TO_i < R_{min}(I^*)$.
  Then there are at least K' faults, where $K' = \lceil (t_i - t_s)/(R_i + TO_i) \rceil$ before t_i. Otherwise, assume that there are $K' - 1$ faults before t_i, then there can be $K - K' + 1$ faults after(or at) t_i, we have $(K - K' + 1)(R_i + TO_i) > (K - (t_i - t_s)/(R_i + TO_i))(R_i + TO_i) \geq RS(J_i) - (t_i - t_s) = d_i' - t_i$, which causes J_i to miss its deadline according to Theorem 1. Since K' faults have already occurred before t_i, this implies that each job in $\mathcal{J}(I^*)$ at least has an additional slack of $K' \times R_{min}(I^*)$ to spare. Since

$$\begin{aligned}K' \times R_{min}(I^*) &= \lceil (t_i - t_s)/(R_i + TO_i) \rceil \times R_{min}(I^*) \\ &\geq (t_i - t_s)/(R_i + TO_i) \times R_{min}(I^*) \\ &\geq t_i - t_s,\end{aligned}$$

  all jobs in $\mathcal{J}(I^*)$ are schedulable.

From the above discussions, we can then conclude that d_i' is a valid deadline for any $J_i \in \mathcal{J}_{I^*}^{do}$.

Case 2: let $J_i \in \mathcal{J}_{I^*}^{ao}$ and a_i' represent the new arrival time, $a_i' = t_s$ and d_i' the updated deadline, i.e. $d_i' = d_i - L(I^*) + RS(J_i)$. Note that J_i has lower priority than all the jobs in $\mathcal{J}(I^*)$. Therefore, we only need to show the changes made to the arrival time and deadline of J_i will not compromise the resource savings to guarantee the schedulability of J_i.

If all the jobs in $\mathcal{J}(I^*)$ and their recoveries finish at or before $t = t_f - RS(J_i)$, then J_i will not experience any interference from jobs in $\mathcal{J}(I^*)$ and its feasibility will not be affected. Now the question becomes what if all jobs in $\mathcal{J}(I^*)$ and their recoveries, if any, finish at t^*, where $t_f - RS(J_i) < t^* \leq t_f$, see Fig. 5b.

We consider the following two cases.

– Case 2-a: $R_i + TO_i >= R_{min}(I^*)$. Then there are at least K' faults, where $K' = \lceil (t^* + RS(J_i) - t_f)/R_{min} \rceil$ before t^*. Otherwise, similar to the proof of Case 1-b, more than $K - K'$ faults occurring at $t = t^*$ will cause at least one job in $\mathcal{J}(I^*)$ to miss its deadline. In the meantime, this implies that J_i at least has an additional slack of $K' \times (R_i + TO_i)$ to spare. We have

$$\begin{aligned} K' \times (R_i + TO_i) &= \lceil (t^* + RS(J_i) - t_f)/R_{min}(I^*) \rceil \times (R_i + TO_i) \\ &\geq t^* + RS(J_i) - t_f. \end{aligned}$$

This ensures that J_i has reserved enough resource for fault recovery.

– Case 2-b: $(R_i + TO_i) < R_{min}(I^*)$. Then there are at most K' faults, where $K' = \lfloor (t_f - t^*)/R_{min}(I^*) \rfloor$ that may occur after t^*. Otherwise, one more fault at t_f will cause some job(s) in $\mathcal{J}(I^*)$ to miss deadline(s). In other words, there must be at least $K - K'$ faults that occurred before t^*. A portion of the shared slacks with the amount of $t^* + RS(J_i) - t_f$ is used by the jobs(recoveries) in $\mathcal{J}(I^*)$. However, job J_i reclaims an additional slack of $(K - K') \times (R_i + TO_i)$ to spare. Since $K \times (R_i + TO_i) \geq RS(J_i)$, we have

$$\begin{aligned} (K - K') \times R_i &\geq (K - \lfloor (t_f - t^*)/R_{min} \rfloor) \times (R_i + TO_i) \\ &\geq (K - (t_f - t^*)/(R_i + TO_i))(R_i + TO_i) \\ &\geq RS(J_i) - (t_f' - t^*)/(R_i + TO_i) \times (R_i + TO_i) \\ &= t^* + RS(J_i) - t_f. \end{aligned}$$

Therefore, J_i also reserves enough resource.

Case 3: let $J_i \in \mathcal{J}_{I^*}^{fo}$. Similarly we want to prove that the change of deadline for J_i will not compromise the resource savings to guarantee its schedulability with the possible of maximum K faults. Since there are $K \times R_{max}(I^*)$ slacks reserved in I^*, it just requires additional slacks of $\max(0, K \times (R_i + TO_i) - K \times R_{max}(I^*))$ for J_i with the sharing mechanism.

We consider two cases below.

– Case 3-a: $(R_i + TO_i) > R_{max}(I^*)$. In this case, additional slacks of $K \times (R_i + TO_i) - K \times R_{max}(I^*)$ is reserved for J_i. Assume that K' faults occurred during the critical interval I^*. J_i can immediately claim the unused reserved slacks of

$(K - K')R_{max}(I^*)$ in I^* to spare. Since there will be at most $K - K'$ faults striking J_i and we have the remaining reserved resources for J_i as

$$\begin{aligned}(K - K')R_{max}(I^*) + (K \times (R_i + TO_i) - K \times R_{max}(I^*)) \\ = K \times (R_i + TO_i) - K'R_{max}(I^*) \\ \geq (K - K')(R_i + TO_i).\end{aligned}$$

This ensures that J_i has reserved enough resources for fault recovery.

– Case 3-b: $(R_i + TO_i) \leq R_{max}(I^*)$. Then there is no additional slacks needed for J_i. Assume that there are K' faults in I^*. This implies J_i can reclaim $(K - K')R_{max}(I^*)$ from the critical interval I^* to spare. In addition, there will be at most $K - K'$ faults affecting J_i. Since $(K - K')R_{max}(I^*) >= (K - K')(R_i + TO_i)$, J_i has enough resources for its execution and recovery.

Since all jobs are associated with a critical interval in LPSSR and all jobs within a critical interval are schedulable when the corresponding speed is applied and the feasibility of the remaining jobs is not affected after the removal of a critical interval, we prove the theorem. □

Algorithm 2 allows reserved slacks to be shared by different critical intervals and thus can achieve better energy efficiency. By far, both EMLPEDF and LPSSR assume that speeds can be continuously varied between $[s_{min}, s_{max}]$. In the next section, we extend our LPSSR algorithm to systems with only a limited number of frequencies.

5 Other considerations of the proposed methods

5.1 Dealing with the limitations of practical processors

Up to now, we assume that the processor speed can be varied continuously. However, current commercial variable voltage processors only have a finite number of speeds Wei et al. (2011), Sridharan and Mahapatra (2010). In addition, it takes time for a processor to change its running modes. These factors must be taken into consideration to provide a practical, valid and efficient voltage schedule.

One intuitive way to deal with discrete frequency levels is to round up the required frequency to the next available level. Unfortunately, this can be extremely pessimistic and energy inefficient, especially for processors with only a few frequencies available. In fact, we can adopt the similar approach as in the work Mochocki et al. (2004) to deal with both the problem of discrete levels of working frequencies and non-zero timing overhead. As shown in Mochocki et al. (2004), non-zero timing overhead can cause monotonicity violation similar to the scenario when we insert recovery blocks for fault tolerance. Therefore, the transition overhead can be efficiently handled by adding it to the reserved blocks. For discrete frequency levels, we can take this factor into consideration when constructing critical intervals. Specifically, when a critical interval is found according to Algorithm 2, its speed needs to be raised to the next level available. Once a higher than necessary speed is used, idle slacks will be generated

in the critical interval. Then we reduce the idle slacks by identifying the latest finishing time of the critical interval.

Given the jobs in the interval and a higher speed than required, we can find the latest finishing time of the workload including recoveries under the worst case as follows. Let $I^* = [t_s, t_f]$ be the critical interval and s_h be its speed. In addition, the set of jobs in I^* is denoted by $\mathcal{J}(I^*)$ and $\mathcal{J}_{hp}(i)$ is the set of jobs of priority higher than that of job J_i. Therefore, the latest finishing time($LFT(I^*)$) is obtained by Eq. (7),

$$LFT(I^*) = \max_{\forall J_i \in \mathcal{J}(I^*)} \left\{ \frac{c_i}{s_h} + \sum_{\forall J_j \in \mathcal{J}_{hp}(i) \cap d_j > a_i} \frac{c_j}{s_h} + K \times Re(i) + a_i \right\} \tag{7}$$

where the first part denotes the execution requirement from J_i itself and the second and third part represent the interference from the higher priority jobs and the worst case recovery time, i.e. $Re(i) = \max\{R_p + TO_p | J_p \in \{J_i\} \cup \mathcal{J}_{hp}(i)\}$ that J_i can suffer, respectively. Note that not all the workload from higher priority jobs are considered because only those with deadlines after the arrival of J_i, i.e. a_i may delay the execution of J_i. Therefore, the actual critical interval to be removed is $[t_s, \min(t_f, LFT(I^*))]$.

To update our LPSSR to deal with discrete frequency levels, we only need to calculate the latest finishing time and update the ending point of the critical interval before line 14. Similarly, this technique can be incorporated into MLPEDF and EMLPEDF as well.

5.2 System reliability and imperfect fault coverage

Our proposed approach enhances the system reliability by ensuring the K-fault-tolerance capability of the system through advanced backup policies. Let the system reliability be defined in 3.

Definition 3 The *reliability* of the system, denoted as R_{sys}, is the probability that the system functions correctly during an operational cycle (its length is represented by L_{cyc}) of the system.

Let $Pr(q, L_{cyc})$ denote the probability that exactly q faults occur during L_{cyc} and ρ denote the fault coverage of the given fault detection method, i.e. $0 < \rho \leq 1$. To ensure the system to function correctly, two conditions have to be met: 1) no more than K faults occur during L_{cyc}; 2) all failures are appropriately detected. As the event of fault occurrence is independent of the process of fault detection, the system reliability R_{sys} can be calculated in Eq. (8),

$$R_{sys} = \sum_{q=1}^{K} Pr(q, L_{cyc}) \cdot Pr_{det}(q), \tag{8}$$

where $Pr_{det}(q) = \rho^q$, i.e. the probability that q faults are detected. If the failure distribution is modeled as a Possion process with a failure rate λ as in Zhao et al. (2004; 2012; 2011), then the reliability function is shown in Eq. (9),

$$R_{sys} = \sum_{q=1}^{K} \frac{(\lambda L_{cyc})^q \cdot e^{-\lambda L_{cyc}}}{q!} \cdot \rho^q \tag{9}$$

As can be seen from both Eqs. (8) and (9), the larger the number of faults, i.e. K that the system can tolerate, the higher the system reliability. Note that, this reliability model is not limited to any particular failure distribution, as long as $Pr(q, L_{cyc})$ is well defined, it can be readily applied. Given a reliability goal and the length of an operational cycle of the system, a corresponding K can be determined under any given fault detection technique.

From Eq. 9, the fault coverage factor can play an important role in system reliability. To study the tradeoffs between different fault coverage techniques is an interesting research problem and will be our future work.

6 Simulation results

In this section, we compare the performance of four algorithms: NPM, MLPEDF, EMLPEDF, and LPSSR. NPM represents the speed schedule with no power management involved, i.e. all jobs or recoveries are executed under s_{max} and is used as a reference schedule. MLPEDF and EMPLEDF are fault tolerant algorithms discussed in Sect. 3 and LPSSR is the algorithm presented in Sect. 4. All energy consumptions plotted were normalized to NPM.

We assumed that $\alpha = 2$, $C_{ef} = 1$, $P_{ind} = 0.05$, and s_{min} was set to 0.25. We tested our algorithm with job sets randomly generated as follows: for each job, the arrival time a_i is uniformly distributed in the interval [0s,100s] while its relative deadline rd_i is in [50s,100s]. Therefore, the absolute deadline was calculated as $d_i = a_i + rd_i$. In addition, the worst case execution time c_i was less than rd_i and also randomly generated. For each job, the timing and energy overhead of fault detection is set to 10 % of its worst case execution time and its energy consumption, respectively. The choices of K were based on the characteristics of the task sets and typical fault arrival rates. As indicated in Zhang and Chakrabarty (2006), the typical fault arrival rate in safety-critical real-time system is in the range of 10^{-10} to $10^{-5}/hour$. However, for systems that operate in harsh environment, the fault arrival rate can be much higher, in the range of 10^{-2} to $10^{2}/hour$. Only the job sets running at s_{max} that are feasible under K faults are of interest to us.

Two sets of simulations were conducted to study the performance of our algorithm in terms of energy savings under continuously varied speeds and discrete speed levels, respectively.

6.1 System with continuous speeds

First, we studied how energy saving performance changes with the number of jobs. We set the fault rate to be 10^{-5} and varied the number of jobs from 10 to 50. For simulations with the same number of jobs, we generated at least 1000 different test cases. With our settings, the number of fault in our job set is no more than 1. Therefore,

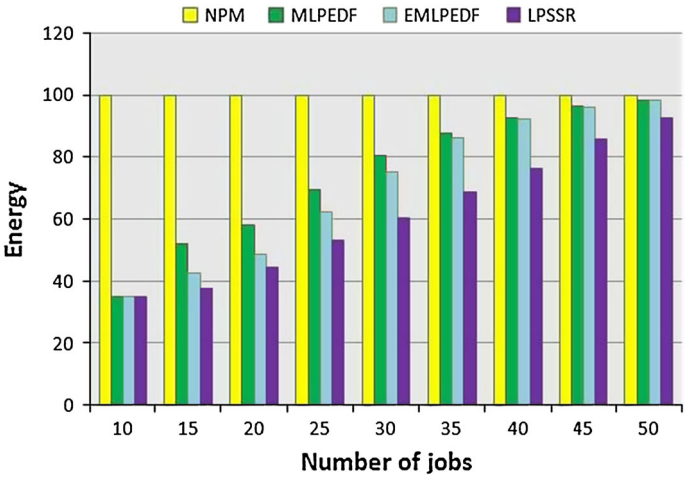


Fig. 6 Energy savings with different numbers of jobs, K = 1

we set $K = 1$. For each job set, we collected the energy consumption of the speed schedule by each of the four approaches. The result is illustrated in Fig. 6.

From Fig. 6, we can see that the energy consumption of LPSSR, EMLPEDF and MLPEDF increases as the job set becomes larger. This is reasonable since the workload is increasing while the slacks that can be used for DVFS are diminishing. LPSSR always dominates the other three algorithms because, by sharing reserved slacks, LPSSR reserves fewer resources for fault recovery and uses more for slowing down the execution of jobs. When the workload is very low, i.e., only 10 jobs, the energy savings achieved by all three algorithms are almost the same, this is due to the fact that most of the test cases are feasible under constant speed s_{min}. When the workload is high enough, most of the slacks are used for fault recovery and no room is left for DVFS. Moreover, if the number of jobs is increased to a certain point, no fault-tolerant speed schedule can be found. In average, additional 13 and 10 % energy saving can be achieved by LPSSR when comparing with MLPEDF and EMLPEDF, respectively.

In our second set of simulations, we wanted to investigate how the number of faults affects the performance of our algorithm. In this simulation, the number of jobs is fixed to 15 and the fault rates to be tolerated varies from 10^{-2} to $10^2/h$, i.e. K changes from 1 to 5. Again, no less than 1,000 different test cases were generated for simulations with the same fault numbers. The average results are shown in Fig. 7.

From Fig. 7 we can see that the energy consumptions by MLPEDF and EMLPEDF increase rapidly as the increase of the number of faults. The energy consumption by LPSSR, on the other hand, grows but less dramatically. From Fig. 7, the energy consumption difference is around 6 % between tolerating 1 fault and 5 faults under LPSSR. This is due to the fact that the recovery slacks are shared to the maximum extent by employing the sharing mechanism in LPSSR. On the contrary, MLPEDF (EMLPEDF) is affected significantly by the increasing number of faults in the system and more than 40 % (33 %) additional energy is consumed when fault occurrences increase from 1 to 5.

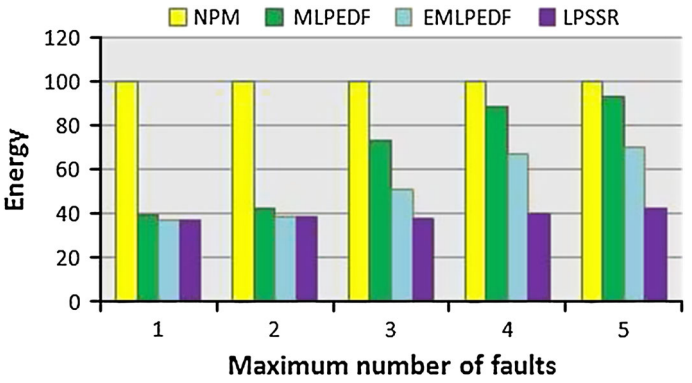


Fig. 7 Energy savings with increasing number of faults, # of jobs = 15

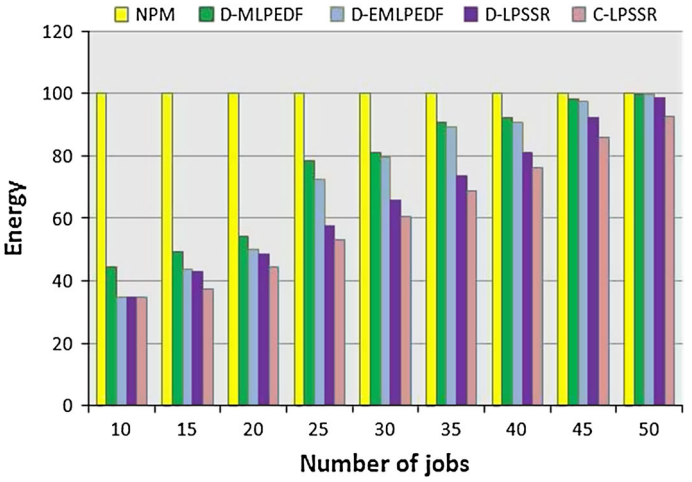


Fig. 8 Energy savings with increasing number of jobs under PentiumM, K = 1

6.2 System with discrete speed levels

In this section, we also evaluate the four algorithms using two different sets of simulations. The technique discussed in Sect. 5.1 is used to deal with discrete speed levels.

We adopt PentiumM processor with 8 frequency levels (1.00, 0.86, 0.76, 0.67, 0.57, 0.47, 0.38, 0.28) as our target system as used in Liu et al. (2010). Two simulations under the same configuration as those in Sect. 6.1 are performed and their results are shown in Figs. 8 and 9, respectively. Again, four algorithms with limited number of speeds are evaluated, which are NPM, D-MLPEDF, D-EMLPEDF and D-LPSSR, respectively. To better illustrate the performance of our algorithm under discrete speed levels, we compare it with that of continuous speeds, which is denoted by C-LPSSR.

The advantages of our algorithm D-LPSSR over the other two in terms of energy savings are manifested in Fig. 8, and the additional energy savings only drops around

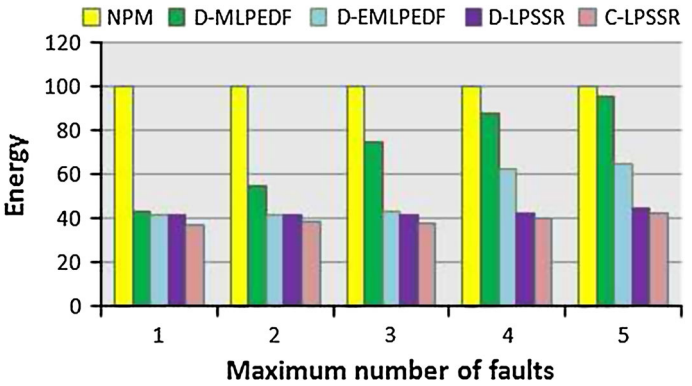


Fig. 9 Energy savings with increasing number of faults under PentiumM, # of jobs = 15

3 % compared with continuous varied speeds. In average, the difference between D-LPSSR and C-LPSSR is only 5 %.

Moreover, for the second simulation, algorithm LPSSR performs even better as shown in Fig. 9. This is due to the fact that it extensively explores the slacks that can be shared among different critical intervals and significantly reduce the amount of recoveries. Therefore, increasing the number of faults has little impact on the resulting speed schedule. Comparing with C-LPSSR, only 3 % more energy is consumed for tolerating 1–5 faults.

6.3 Real-life periodic task sets

In this section, we verify the proposed algorithms using three real-life periodic task sets, which are a CNC task set, an inertial navigation system (INS) task set, and a generic aviation platform (GAP) task set, respectively. The specifications of these task sets can be found in Zhang and Chakrabarty (2006) and omitted here due to space limitation. Based on our simulations, no task set can tolerate more than 2 faults. Therefore, only the results of K = 1, 2 are recorded and are normalized to NPM. Processors with continuous frequencies and discrete frequency levels are considered, separately.

As shown in Table 2, all three algorithms can achieve energy savings compared with NPM while maintaining the feasibility of the task sets, where A1, A2 and A3 stand for MLPEDF, EMLPEDF and LPSSR, respectively. The two algorithms, i.e. EMLPEDF and LPSSR have similar performance when tolerating 1 fault, because the shared slacks are negligible considering a relatively small execution time to period ratio. For all three algorithms, we noticed a consequent increase in energy consumption when K increases. This increase mostly comes from the first iteration of the algorithm, the intensity of the first critical interval is much higher for a larger K, especially for a task set with large utilization where slacks are already scarce. However, when K = 2, LPSSR stills attains another 8.5 % (12 %) energy reduction compared with EMLPEDF (MLPEDF), which is a strong demonstration of the benefits from slack-

Table 2 Energy-performance comparison for CNC, INS, and GAP

		Continuous frequencies			PentiumM		
Task set	K	A1 (%)	A2 (%)	A3 (%)	A1 (%)	A2 (%)	A3 (%)
CNC	1	69.9	60.4	59.9	72.8	62.6	61.9
	2	86.4	80.5	71.4	91.8	84.8	77.4
INS	1	96.3	93.0	88.5	98.7	96.2	92.1
	2	NF	NF	NF	NF	NF	NF
GAP	1	91.5	89.4	87.2	98.4	96.9	93.3
	2	100	100	92.2	100	100	96.8

sharing. Under a processor with a limited number of frequencies, i.e. PentiumM in Sect. 6.2, the performance of our algorithms is slightly degraded as expected.

6.4 Further validation of LPSSR

To our best knowledge, there is no other paper in the literature addressing the exactly same problem. However, to demonstrate the efficacy of our LPSSR, we compared LPSSR against the method fault-tolerant uniform checkpointing with DVFS (FTUniChK) from the work Melhem et al. (2004) that studied the fault-tolerant energy reduction for periodic task sets scheduled under EDF on a single processor. FTUniChK first identified the the checkpointing interval and then derived a constant speed to execute the entire task set, but it is only applicable when no more than one fault can occur, i.e. $K = 1$. Note that, our LPSSR exploits the slacks that can be shared among different jobs and acts on top of any checkpointing scheme. Therefore, we directly adopted the uniform checkpointing scheme from Melhem et al. (2004) before employing LPSSR.

The simulation parameters were set as follows. We had $\alpha = 2, C_{ef} = 1, P_{ind} = 0.05$, and s_{min} was set to 0.25. Each task set consisted of 10 periodic tasks, whose periods were uniformly generated in the range of [5s 50s]. The checkpointing overhead of each task was set to 5 % of its worst case execution time under s_{max}. The total utilization of the task set was varied from 0.2 to 0.95 with a step of 0.05. For each utilization value, we generated 1000 different task sets according to UUNISORT in Bini and Buttazzo (2005), and the average energy consumption of one LCM was reported. We again normalized the energy consumption with respect to that of NPM.

According to Fig. 10, our LPSSR consistently outperforms FTUniChK. When the processor is light-loaded, both methods use close-to-minimum speed to execute the task set, therefore the energy performance is close. However, as the utilization increase, LPSSR can reduce the amount of slacks reserved for fault-tolerance and use more for energy reduction compared to FTUniChK. For instance, when the utilization is 0.8, LPSSR achieves around 10 % more energy savings.

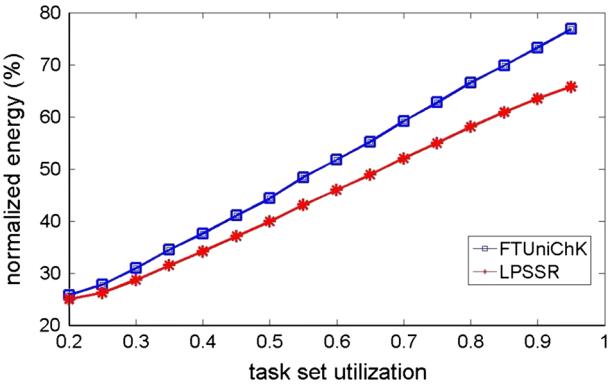


Fig. 10 LPSSR versus FTUniCk

Through extensive simulations, we have shown that the three proposed algorithms can save a significant amount of energy comparing with NPM. Specifically, our LPSSR algorithm is more energy efficient by reserving the least amount of slacks.

7 Conclusion

In this paper, we investigate the problem of minimizing energy consumption when scheduling a set of real-time jobs in presence of up to K transient faults under EDF policy. We explore the reserved slacks in the system and maximize its utility by providing a slack sharing mechanism. Under the notion of shared recovery slacks, we propose an algorithm that reduces the energy consumption and maintains feasibility under the worst case, i.e. up to K faults occur during one operational cycle of the system. We then extend our algorithm to systems with discrete speed levels to provide practical and energy efficient solutions. Theoretical validation of our approach is provided and the simulation results have shown that our approach consistently results in lower energy consumption compared with other algorithms.

Acknowledgments This work is supported in part by NSF under Projects CNS-1423137 and CNS-1018108.

References

AMD (2014) Amd g-series. http://www.amd.com/us/products/embedded/processors/Pages/g-series.aspx. Accessed 20 Feb 2014

Aydin H (2007) Exact fault-sensitive feasibility analysis of real-time tasks. IEEE Trans Comput 56(10):1372–1386. doi:10.1109/TC.2007.70739

Aydin H, Melhem R, Mosse D, Mejia-Alvarez P (2004) Power-aware scheduling for periodic real-time tasks. IEEE Trans Comput 53(5):584–600. doi:10.1109/TC.2004.1275298

Bini E, Buttazzo GC (2005) Measuring the performance of schedulability tests. Real-Time Syst 30(1–2):129–154. doi:10.1007/s11241-005-0507-9

Castillo X, McConnel SR, Siewiorek DP (1982) Derivation and calibration of a transient error reliability model. IEEE Trans Comput 31:658–671. doi:10.1109/TC.1982.1676063

Davis RI, Burns A (2007) Controller area network (can) schedulability analysis: refuted, revisited and revised. refuted, revisited and revised. Real-Time Syst 35:239–272

Intel (2012) Intel xeon processor. http://www.intel.com/content/www/us/en/intelligent-systems/crystal-forest-server/xeon-e5-v2-89xx-chipset.html. Accessed 20 Feb 2014

Izosimov V, Pop P, Eles P, Peng Z (2012) Scheduling and optimization of fault-tolerant embedded systems with transparency/performance trade-offs. ACM Trans Embed Comput Syst 11(3):61:1–61:35. doi:10.1145/2345770.2345773

Langley T, Koga R, Morris T (2003) Single-event effects test results of 512mb sdrams. In: Radiation effects data workshop, IEEE, pp 98–101. doi:10.1109/REDW.2003.1281355

Lawrence R (2007) Radiation characterization of 512mb sdrams. In: Radiation effects data workshop, vol 0, IEEE, pp 204–207. doi:10.1109/REDW.2007.4342566

Liu Y, Liang H, Wu K (2010) Scheduling for energy efficiency and fault tolerance in hard real-time systems. In: Design, automation test in Europe conference exhibition (DATE), pp 1444–1449

Many F, Doose D (2011) Scheduling analysis under fault bursts. In: Real-Time and embedded technology and applications symposium, IEEE, pp 113–122. doi:10.1109/RTAS.2011.19

Melhem R, Mosse D, Elnozahy E (2004) The interplay of power management and fault recovery in real-time systems. IEEE Trans Comput 53(2):217–231. doi:10.1109/TC.2004.1261830

Mochocki B, Hu X, Quan G (2004) A unified approach to variable voltage scheduling for nonideal dvs processors. IEEE Trans Comput-Aided Des Integr Circuits Syst 23(9):1370–1377. doi:10.1109/TCAD.2004.833602

Pop P, Poulsen KH, Izosimov V, Eles P (2007) Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis, CODES+ISSS '07, ACM, New York, pp 233–238. doi:10.1145/1289816.1289873

Pradhan DK (ed) (1996) Fault-tolerant computer system design. Prentice-Hall Inc, Upper Saddle River

Quan G, Hu X (2001) Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In: DAC '01, Proceedings of the 38th annual design automation conference, ACM, New York, pp 828–833. doi:10.1145/378239.379074

Quan G, Niu L (2004) Fixed priority scheduling for reducing overall energy on variable voltage processors. In: IEEE computer Society in 25th IEEE Real-Time system symposium, IEEE, pp 309–318

Skotnicki T, Hutchby J, King TJ, Wong HS, Boeuf F (2005) The end of cmos scaling: toward the introduction of new materials and structural changes to improve mosfet performance. IEEE Circuits Dev Mag 21(1):16–26. doi:10.1109/MCD.2005.1388765

Sridharan R, Mahapatra R (2010) Reliability aware power management for dual-processor real-time embedded systems. In: 47th ACM/IEEE design automation conference (DAC), IEEE, pp 819–824

Srinivasan J, Adve S, Bose P, Rivers J (2004) The impact of technology scaling on lifetime reliability. In: 2004 International conference on dependable systems and networks, pp 177–186. doi:10.1109/DSN.2004.1311888

Srinivasan J, Adve SV, Bose P, Rivers J, Hu CK (2003) Ramp: A model for reliability aware microprocessor design. IBM Research Report, RC23048. New York

Wei T, Chen X, Hu S (2011) Reliability-driven energy-efficient task scheduling for multiprocessor real-time systems. IEEE Trans Comput-Aided Des Integr Circuits Syst 30(10):1569–1573. doi:10.1109/TCAD.2011.2160178

Wei T, Mishra P, Wu K, Zhou J (2012) Quasi-static fault-tolerant scheduling schemes for energy-efficient hard real-time systems. J Syst Softw 85(6):1386–1399. doi:10.1016/j.jss.2012.01.020

Yao F, Demers A, Shenker S (1995) A scheduling model for reduced cpu energy. In: Proceedings of 36th annual symposium on foundations of computer science, pp 374–382. doi:10.1109/SFCS.1995.492493

Zhang Y, Chakrabarty K (2006) A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems. IEEE Trans Comput-Aided Des Integr Circuits Syst 25(1):111–125. doi:10.1109/TCAD.2005.852657

Zhang Y, Chakrabarty K, Swaminathan V (2003) Energy-aware fault tolerance in fixed-priority real-time embedded systems. In: ICCAD '03, Proceedings of the 2003 IEEE/ACM international conference on Computer-aided design, IEEE Computer Society, Washington, DC, p 209. doi:10.1109/ICCAD.2003.63

Zhao B, Aydin H, Zhu D (2011) Generalized reliability-oriented energy management for real-time embedded applications. In: 48th ACM/EDAC/IEEE design automation conference (DAC), IEEE, pp 381–386

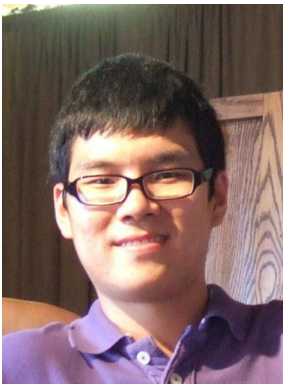
Zhao B, Aydin H, Zhu D (2012) Energy management under general task-level reliability constraints. In: IEEE 18th real-time and embedded technology and applications symposium (RTAS), pp 285–294. doi:10.1109/RTAS.2012.30

Zhu D, Melhem R, Mosse D (2004) The effects of energy management on reliability in real-time embedded systems. In: ICCAD '04, Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design, IEEE Computer Society, Washington, DC, pp 35–40. doi:10.1109/ICCAD.2004.1382539

Qiushi Han is a Ph.D. candidate in the Department of Electrical and Computer Engineering at the Florida International University, Florida, USA. He received his B.S. from the Department of Software Engineering, Beijing Jiaotong University. His research interests include real-time systems, power-/thermal- aware computing and reliable/fault-tolerant system designs.

Linwei Niu received the B.S. in Computer Science and Technology from Peking University, Beijing, China in 1998, the M.S. in Computer Science from State University of New York at Stony Brook in 2001, and the Ph.D. in Computer Science and Engineering from University of South Carolina in 2006. Currently, he is an associate professor in the Department of Math and Computer Science, West Virginia State University, U.S.A. His research interests include power-aware design for embedded systems, design automation, real-time scheduling and software/hardware co-design.

Gang Quan received his Ph.D. from the Department of Computer Science & Engineering, University of Notre Dame, USA, his M.S. from the Chinese Academy of Sciences, Beijing, China, and his B.S. from the Department of Electronic Engineering, Tsinghua University, Beijing, China. He is currently an associate professor in the Electrical and Computer Engineering Department, Florida International University. Before he joined the department, he was an assistant professor at the Department of Computer Science and Engineering, University of South Carolina. His research interests and expertise include real-time systems, embedded system design, power-/thermal-aware computing, advanced computer architecture and reconfigurable computing. Dr. Quan is the recipient of a National Science Foundation Faculty Career Award. He also won the Best Paper Award from the 38th Design Automation Conference. His paper was also selected as one of the Most Influential Papers of 10 Years Design, Automation, and Test in Europe Conference (DATE) in 2007. Dr. Quan is a senior member of IEEE.



Shaolei Ren received the B.E. degree in Electronic Engineering from Tsinghua University in 2006, the M.Phil. degree in Eletronic and Computer Engineering from Hong Kong University of Science and Technology in 2008, and the Ph.D. degree in Electrical Engineering from University of California, Los Angeles, in 2012. Since August 2012, he has been with Florida International University, where he currently holds a joint appointment of Assistant Professor in the School of Computing and Information Sciences and the Department of Electrical and Computer Engineering. His research interests lie in cloud computing, data center resource management and network economics.

Shangping Ren is an associate professor in the Computer Science Department at the Illinois Institute of Technology. She earned her Ph.D from UIUC in 1997. Before she joined IIT in 2003, she worked in software and telecommunication companies as a software engineer and then lead software engineer. Her current research interests include coordination models for real-time distributed open systems, real-time, fault-tolerant and adaptive systems, Cyber-Physical System, parallel and distributed systems, cloud computing, and application-aware many-core virtualization for embedded and real-time applications.