Energy Management under General Task-Level Reliability Constraints

Baoxian Zhao, Hakan Aydin
Department of Computer Science
George Mason University
Fairfax, VA 22030
bzhao@gmu.edu, aydin@cs.gmu.edu

Dakai Zhu
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
dzhu@cs.utsa.edu

***Abstract*—The negative impact of the popular energy management technique Dynamic Voltage and Frequency Scaling (DVFS) on the reliability of real-time embedded systems, in terms of increased transient fault rates, has been recently identified. As a result, recent research literature includes a number of solutions within the so-called Reliability-Aware Power Management (RA-PM) framework, where the aim is to preserve the system's original reliability. In this research effort, we propose a more general framework where the aim is to achieve arbitrary reliability levels that may vary for each periodic task. A critical component of our solution is the use of dynamically allocated recoveries: we show that providing a relatively modest recovery allowance to a given periodic task helps to achieve surprisingly high reliability levels as long as these allowances can be reclaimed on-demand during the hyperperiod. We propose a pseudo-polynomial time feasibility test, as well as static and dynamic algorithms to determine the recovery allowance and frequency assignments to minimize energy consumption while satisfying timing and reliability constraints. Our experimental evaluation points to the significant gain potential of the new framework in terms of both energy and reliability figures.**

I. Introduction

Many of the state-of-the-art energy management frameworks for real-time embedded systems employ the *Dynamic Voltage and Frequency Scaling (DVFS)* technique. With DVFS, the supply voltage and frequency of the processor are simultaneously scaled at run-time to save power at the expense of increased task response times. Several processor technologies, including Intel *SpeedStep* and AMD *PowerNow!*, have direct support for DVFS. The energy-aware scheduling of real-time tasks in the presence of DVFS has been extensively studied in the last decade [3], [4], [9], [19].

On the other hand, the *reliability*, and in general, *fault tolerance* objectives are of paramount importance for real-time systems: faults that can occur at run-time can cause errors and/or deadline misses. While permanent faults can bring the entire processing unit to a halt (and cannot be tolerated without spare computing units), it is shown that most of the faults are caused by environmental factors (such as electromagnetic interference or cosmic rays [34]) and they affect the systems for short durations. These *transient* faults can result in soft errors and erroneous computations, affecting tasks in execution. On single-processor systems, transient faults can be tolerated by the *backward recovery* techniques that typically rely on *time redundancy* [1], [21], [31]: faulty computation can be repeated if there is sufficient time before the task's deadline.

The three-dimensional interplay of energy, reliability, and timeliness objectives introduces non-trivial challenges. First, both DVFS and time-redundancy based recovery techniques actively compete for the use of available system slack that can be used either for slow-down or as reserved recovery slots. Second, recent studies revealed that DVFS comes at the cost of significantly increased transient fault rates [25], [32], suggesting that provisioning for run-time faults becomes even more important in DVFS settings. The *reliability-aware power management (RA-PM)* framework [30], [31], [33] is proposed to address the reliability-degradation problem associated with DVFS. Specifically, RA-PM solutions allocate a recovery job to every real-time job whose frequency is scaled down, before its deadline. In this way the recovery can be executed, should an error be detected at the end of job's execution. It is shown that this approach preserves the *original reliability* of the task set, which is defined as the probability of completing all the jobs with success when voltage/frequency scaling is disabled [30], [31].

The main objectives of this research effort are twofold:

1. To lay the foundations of a more comprehensive framework to achieve *arbitrary* reliability levels for *individual* periodic tasks, when employing DVFS. This may prove very useful for applications with different/mixed criticality (or, importance) levels whose requirements may not be fully captured by simply preserving the original reliability levels. For instance, some critical tasks may require very high reliability levels – in fact, reliability levels that require the use of recoveries even when not using DVFS may be sought. Conversely, for some other tasks, a modest reliability reduction may be acceptable in exchange for high energy savings. *Such task-variant reliability objectives can neither be expressed nor achieved in existing RA-PM solutions.*

2. To investigate and exploit the potential of deploying *dynamically allocated recoveries* for periodic tasks, in co-management of reliability and energy. As opposed to the current RA-PM schemes that *statically* allocate a *separate*

recovery to each and every scaled job, our framework is based on providing every periodic task with a certain *recovery allowance* for the execution. The recoveries can be used by any number of the jobs of the task under consideration during the hyperperiod, as long as the recovery allowance is not exceeded. Our analysis and results indicate that such a dynamic recovery allocation strategy is highly effective, in the sense that: *i*. even with rather small number of recovery allowances, a surprisingly high reliability levels can be obtained, and, *ii*. energy savings can be significantly improved due to the less conservative and task-dependent recovery reservation. Eventually, these two leverage dimensions help us to formulate and tackle the general problem of **determining recovery allowance and frequency assignments to minimize energy consumption, while meeting the timing constraints and task-level reliability targets**.

The remaining of this paper is organized as follows. After presenting our models and basic definitions in Section II, we compare existing recovery strategies and illustrate the principles, as well as the potential of, dynamic allocation of recoveries through concrete examples in Section III. In Section IV, we discuss the main dimensions of the general problem, which are subsequently addressed in Section V (the feasibility problem) and Section VI (the frequency and recovery allowance assignment problem). Section VI presents our two proposed schemes as well as their dynamic extensions, and includes their detailed experimental comparison. Finally, Section VII gives the closely related work and Section VIII concludes the paper.

II. Models and Definitions

A. Task Model

We consider a set of independent periodic real-time tasks $\Gamma = \{T_1, .., T_n\}$. Each task T_i is characterized by a period p_i, its *worst-case (WC)* execution time c_i, and the *best-case (BC)* execution time bc_i. As we consider DVFS-enabled processors, these execution times correspond to the cases where tasks are executed at the maximum processor speed. The relative deadline of task T_i is assumed to be equal to its period. The j^{th} job of T_i, denoted as T_{ij}, arrives at time $(j-1)\cdot p_i$ and has a deadline of $j \cdot p_i$. We define the *hyperperiod* H of the task set as the least common multiple (LCM) of all tasks' periods. The total number of jobs of task T_i during the hyperperiod is represented by $k_i = \frac{H}{p_i}$.

We assume that the DVFS-enabled processor has ℓ discrete speeds levels, $s_{min} = s_1 < s_2 \ldots < s_\ell = s_{max}$. Here, $s_{min} = s_1$ stands for the minimum available speed of the processor. Moreover, for simplicity, we normalize speed levels with respect to the maximum speed s_{max}, where $s_{max} = s_\ell = 1.0$. Note that, in modern processors, ℓ is typically a small number not exceeding 10.

The nominal utilization of task T_i under s_{max} is defined as $u_i = \frac{c_i}{p_i}$. The (nominal) system utilization is further defined as $U = \sum_{i=1}^{n} u_i$. Assuming preemptive Earliest-Deadline-First (EDF) scheduling, the necessary and sufficient condition for feasibility under the maximum speed s_{max} is $U \leq 1.0$ [18]. The execution frequency (speed) of task T_i is denoted by f_i. Clearly, f_i can assume only one of the discrete speed levels in $\{s_1, \ldots, s_\ell\}$. We assume that the task may take up to $\frac{c_i}{f_i}$ time units when executed at frequency f_i.

B. Energy Model

Considering the increasing static power trends with scaled feature sizes, as well as the existence of multiple system components consuming power, it has been observed that power management schemes that focus on individual components may not be energy efficient at the system level and system-wide power management becomes a necessity [3], [16]. In this paper, we adopt a simple system-level power model, where the power consumption of a system running at speed s can be expressed as [3], [32]:

$$P(s) = P_s + \hbar(P_{ind} + P_d) = P_s + \hbar(P_{ind} + C_{ef} \cdot s^m) \quad (1)$$

Above, P_s stands for *static power*, which can be removed only by powering off the whole system. Due to the prohibitive overhead of turning off/on a system in periodic real-time execution settings, we assume that the system is in *on* state at all times and that P_s is always consumed. Hence, we will focus on the energy consumption related to active power, represented by the second component in the above equation. The coefficient $\hbar$ is 1 when the system actively executes a task; otherwise, $\hbar = 0$. P_{ind} stands for the *frequency-independent active power*, which includes any active power that does not depend on running speed and can be effectively removed by putting the system to sleep. P_{ind} is assumed to be a constant. The *frequency-dependent active power* P_d depends on the system running speed s, and system-dependent constants C_{ef} and m [6]. From this model, one can derive the minimum *energy-efficient speed* value as $s_{ee} = \sqrt[m]{\frac{P_{ind}}{C_{ef}\cdot(m-1)}}$ [3], [32]. That is, for energy efficiency, no job should be executed at a speed lower than s_{ee} as doing so would result in higher energy consumption.

C. Fault and Reliability Models

During the operation of a computing system, both *permanent* and *transient* faults may occur due to, for instance, the effects of hardware defects or cosmic ray radiations, which can result in system *errors*. Transient faults, which are the focus of this paper, have been shown to be dominant [14] especially with scaled technology sizes [13]. Transient faults have been traditionally modeled by Poisson distributions, where the average arrival rate of soft errors caused by such faults is assumed to be λ [25]. However, considering the negative effects of DVFS on transient faults, soft error rates at a lower speed s ($< s_{max}$) (and the corresponding supply voltage V) can be modeled as [32]:

$$\lambda(s) = \lambda_0 \cdot 10^{\frac{d \cdot (1-s)}{1-s_{min}}} \quad (2)$$

where λ_0 corresponds to the average error rate at the maximum speed s_{max} and d (> 0) is a constant, which represents the sensitivity of soft errors caused by transient faults due to DVFS. That is, reducing the supply voltage and processing speed for energy savings can lead to exponentially increased soft errors [8], [32]. The maximum average arrival rate of soft errors is assumed to be $\lambda_{max} = \lambda_0 \cdot 10^d$, which corresponds to the lowest processor speed s_{min} (and minimum supply voltage V_{min}) [31], [32].

The reliability of *a single job* of task T_i, running at frequency f_i, is the probability of completing the job without incurring errors due to transient faults and is given as [31], [32]:

$$R_i(f_i) = e^{-\lambda(f_i) \cdot \frac{c_i}{f_i}} \quad (3)$$

where c_i is the worst-case execution time of T_i under the maximum processor speed. The *original* reliability of a single job of T_i, denoted by R_i^0, is the one that corresponds to the case where the job runs at the maximum processing frequency. That is, $R_i^0 = R_i(1.0)$.

Definition 1. *The* task-level reliability *of task T_i, denoted by Φ_i, is the probability of completing all k_i instances of T_i successfully during a hyperperiod H.*

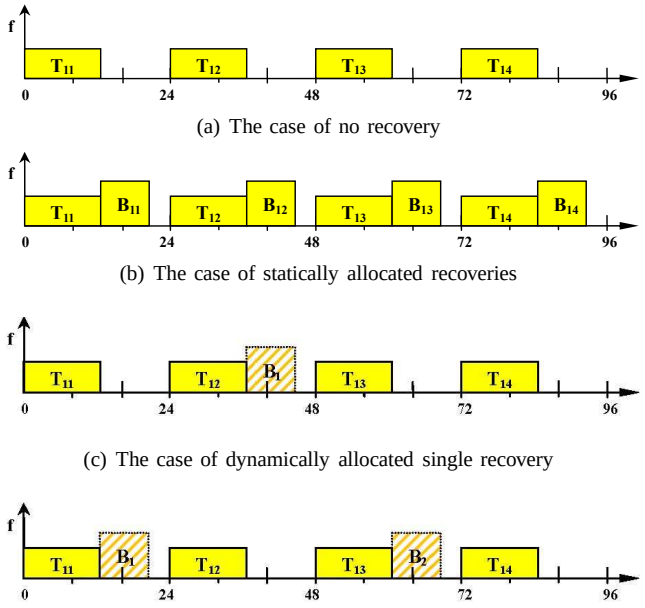
Definition 2. *The system's overall reliability, denoted by Φ_{sys}, is defined as completing all jobs successfully during a hyperperiod H, and is given by $\Phi_{sys} = \prod_{i=1}^{n} \Phi_i$.*

It can be seen that task-level and system-level reliabilities will, among other factors, highly depend on the running frequency assignment to tasks. The task-level and system-level *original* reliabilities are defined as the ones that result from running all jobs at the maximum speed during a hyperperiod [31], which are denoted by Φ_i^0 and Φ_{sys}^0, respectively. Finally, the *target* reliability of task T_i is denoted by Φ_i^t.

III. Recovery Strategies for Periodic Execution Model

To recover from the soft errors triggered by the transient faults, we can exploit backward recovery technique and improve task/system's reliability. In this section, we compare the impact of different recovery allocation strategies on the reliability of periodic tasks. Consider the case where the task T_i runs at the frequency f_i during hyperperiod H. The exact expression of its task-level reliability Φ_i will depend on the *number* and *distribution* of recovery tasks.

- **Case 1: No recoveries.** In this case, no provisions are made to recover from potential transient faults that can affect individual jobs. As a result, the entire execution will be successful if and only if there are no errors induced by transient faults during the hyperperiod. We find $\Phi_i = (R_i(f_i))^{k_i}$, and further, $\Phi_{sys} = \prod_{i=1}^{n} (R_i(f_i))^{k_i}$, which corresponds to the probability of completing all job instances without incurring any transient faults during the hyperperiod. This approach has the clear drawback of reducing the reliability by great margins. As a concrete example, consider the task T_1 with worst-case execution time $c_1 = 8ms$ and the period $p_1 = 24ms$. Assume the hyperperiod includes $k_i = 4$ job instances of T_1, namely, T_{11}, T_{12}, T_{13} and T_{14}. The transient fault model uses the parameters from [31] with $\lambda_0 = 10^{-6}$, $d = 3$ and $s_{min} = 0.1$. When all the jobs run at the maximum frequency (1.0), the overall probability of failure (*PoF*, defined as *1 - Reliability*), is evaluated as 8×10^{-9}. If we scale down all these jobs to a low frequency $f = 0.6$ as shown in Figure 1(a), the new *PoF* of the task is found as 1.15×10^{-5}. Observe how the lack of provision for recoveries results in a reliability degradation by more than four orders of magnitude even for a single task, during the hyperperiod.



(a) The case of no recovery

(b) The case of statically allocated recoveries

(c) The case of dynamically allocated single recovery

(d) The case of dynamically allocated dual recoveries

Figure 1. Impact of recovery strategies for a task running at the scaled frequency $f = 0.6$ over $k_i = 4$ consecutive instances

- **Case 2: Statically allocated recoveries.** Now, consider the case where the recovery jobs are assigned *statically* to a subset of jobs of the periodic task, while the remaining jobs run without relying on a recovery. Specifically, if a recovery job is assigned statically to one of the instances of T_i, then the recovery is executed at the maximum frequency if a fault is detected at the completion time of that specific instance. As a result, the probability of successfully completing that single

instance (in other words, its new reliability) is [30]:

$$R''(f_i) = R_i(f_i) + R'_i(f_i)$$

where $R'_i(f_i) = (1 - R_i(f_i))R_i(1.0)$. Above, the first component $R_i(f_i)$ corresponds to the probability of completing the job without any transient fault, while the second component $R'(f_i)$ indicates the probability of having a transient fault, which is later successfuly recovered from by re-executing at the maximum normalized frequency 1.0. Since $R_i(1.0) = R_i^0$ by definition, $R''_i(f_i)$ is known to be no less than the job's original reliability R_i^0 [30].

For T_i's task-level reliability, if b_i instances have *statically* allocated recoveries and $k_i - b_i$ instances run without any recovery provision, we have:

$$\Phi_i = (R''_i(f_i))^{b_i} \times R_i(f_i)^{k_i - b_i}$$

A special case warrants further elaboration: All jobs of T_i are scaled and $b_i = k_i$. This corresponds to the traditional RA-PM solutions [31] and yields:

$$\Phi_i = \Phi_{RAPM,i} = (R''_i(f_i))^{k_i}$$

Obviously, Φ_i is maximized in this approach and the *scaled* task's reliability is guaranteed to be better than its original reliability Φ_i^0.

If we return to our running example, in order to maintain the system original reliability, the existing RA-PM scheme [31] will statically schedule a recovery job B_{1j} for each individual job instance T_{1j} during the hyperperiod as shown in Figure 1(b), and the new *PoF* is found as 9.19×10^{-13} which is now *better* than the task's original reliability by approximately four orders of magnitude.

However, allocating a separate recovery to every job instance of a scaled task requires significant amount of static slack and affects the energy savings opportunities of *other tasks* negatively. Hence, a significant number of tasks may remain *un-managed* (i.e., may have to run at the maximum frequency without any recovery) [31]. Also note that RA-PM is in general unable to target a specific reliability level which may be higher or lower than the task's original reliability.

- **Case 3: Dynamically allocated recoveries.** Another possibility, which is the proposal of this research effort, is to provide each task with a certain *recovery allowance* for execution.

  Specifically, provisions will be made through a static analysis to provide up to $a_i \leq k_i$ recoveries to task T_i *anywhere in the hyperperiod*, without associating a given recovery with a specific task instance in advance. The net result is that, at runtime, the task will be able to use these dynamically allocated recoveries for any a_i *arbitrary* instances, effectively covering $\sum_{j=1}^{a_i} \binom{k_i}{j}$ distinct fault scenarios for task T_i.

  To illustrate these points, consider task T_i running with a single ($a_i = 1$) dynamic recovery over k_i instances during the hyperperiod H. The execution will be successful if:
  - No task instance encounters a fault, or,
  - The j^{th} instance encounters a fault, the single dynamic recovery is successfully executed and the job instances except T_{ij} complete without encountering a transient fault, for every $j = 1, \ldots, k_i$.

  This effectively gives a reliability figure of

$$\Phi_i = R_i(f_i)^{k_i} + k_i R'_i(f_i) R_i(f_i)^{k_i - 1}$$

The reader should observe how even a single recovery allowance effectively provisions for $k_i + 1$ different execution scenarios. Moreover, due to typically low transient fault rates, these typically cover the scenarios with *maximum probability of occurrence*. In other words, the probability of having fault scenarios with increasing number of faults affecting multiple instances of the *same* task, while not exactly zero, will quickly drop to very small numbers.

Increasing the level of dynamic recovery allowance to $a_i = 2$ would provision for cases where faults affect any two arbitrary instances, covering an additional $\frac{k_i(k_i-1)}{2}$ fault scenarios. Therefore, we have:

$$\Phi_i = R_i(f_i)^{k_i} + k_i R'_i(f_i) R_i(f_i)^{k_i - 1} + \frac{k_i(k_i - 1)}{2} (R'_i(f_i))^2 R_i(f_i)^{k_i - 2}$$

In general, for a task T_i running at speed f_i and with a_i recovery allowances for k_i job instances during H, the task-level reliability $\Phi_i = \Phi_i(f_i, a_i, k_i)$ is found as:

$$R_i(f_i)^{k_i} + \sum_{j=1}^{a_i} \binom{k_i}{j} (R'_i(f_i))^j R_i(f_i)^{k_i - j} \qquad (4)$$

In our running example, if we give a single recovery allowance that can be used by any of the four scaled down job instances (Figure 1(c)) the new *PoF* is evaluated as 6×10^{-11}. In the figure, the dashed lines around the recovery task B_1 indicate that it is not statically associated by any specific job, but can be dynamically scheduled whenever a transient fault is detected in any job. Clearly, with the new *PoF* in this example, we still preserve, and in fact improve by two orders of magnitude, the task's original reliability. Further, by dynamically scheduling $a_1 = 2$ recoveries for these job instances, the *PoF* reaches a level of 9.20×10^{-13}, which is extremely close to that obtained by the RA-PM technique [31]. This simple example illustrates that dynamic allocation of recoveries can also achieve very high reliability levels (comparable to RA-PM) by reserving smaller number of recovery slots.

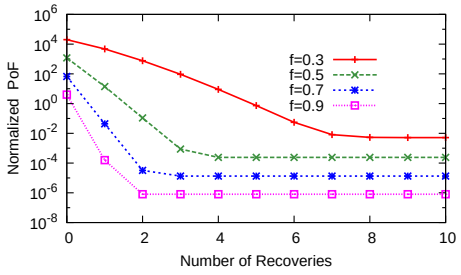


Figure 2. $d = 5$: Impact of number of recoveries on *PoF* for different frequencies

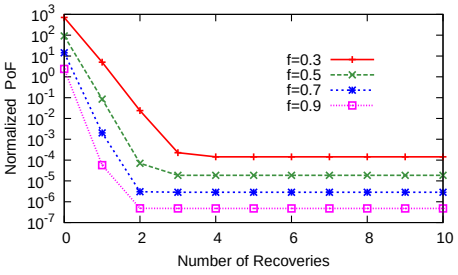


Figure 3. $d = 3$: Impact of number of recoveries on *PoF* for different frequencies

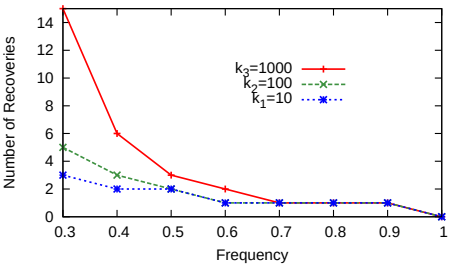


Figure 4. Impact of frequency on the number of recoveries needed to maintain Φ^0

We now elaborate further on how the reliability levels of a single periodic task T_i change with the recovery allowance a_i, frequency f_i, and the number of jobs k_i within the hyperperiod H.

Consider a periodic task T_i, with $c_i = 20ms$, $p_i = 100ms$ and $k_i = 100$ instances within a hyperperiod. We further assume that $\lambda_0 = 10^{-6}$ and the fault-sensitivity exponent d may vary from 3 to 5 [32]. Figures 2 and 3 show the relationship between the recovery allowance a_i and the achieved *PoF* for the cases of $d = 5$ and $d = 3$, respectively. Here, we report *PoF* values normalized with respect to the one that corresponds to the task's original reliability (i.e., $1 - \Phi_i^0$). As we can see, for a given frequency level, increasing the number of recoveries significantly improves the reliability (reduces the *PoF*). However, after the recovery allowance reaches a certain number for a given frequency, the extra reliability gain becomes extremely small, in fact negligible – because the likelihood of new fault scenarios that can be covered by larger recovery allowances approaches quickly zero, thanks to the very nature of *dynamic allocation* strategy. Also, as expected, low frequency levels typically need more recoveries to achieve a given *PoF*. Reducing the fault-sensitivity exponent d from 5 (Figure 2) to 3 (Figure 3) makes the transient faults less likely, and the *PoF* values further drop with the same recovery allowance.

Figure 4 shows the number of recoveries needed to maintain the task-level original reliability as a function of the frequency f, for various k_i values (the number of jobs within the hyperperiod) and $d = 5$. In general, the recovery allowance needed to maintain Φ_i^0 drops with increasing frequency and decreasing k_i (a measure of the length of the hyperperiod). An interesting observation from Fig. 4 is that with the dynamic allocation of only a small number of recovery allowances, the task's original reliability can be maintained. In general, the same observation holds for arbitrary task-level reliability targets, underlining the potential of the new technique compared to the traditional RA-PM.

IV. Dimensions of the Problem

The dynamically allocated recovery framework gives a powerful tool to enhance task-level reliabilities of periodic tasks with minimum recovery allowance. However, the ultimate design problem we aim to address is the following: **Given a set of periodic tasks with task-level reliability objectives $\{{\Phi_i}^t | \ i = 1, \ldots n\}$, how to choose frequency $\{f_i\}$ and recovery allowance $\{a_i\}$ assignments to minimize energy consumption?** Consideration of this problem mandates, in the first place, addressing the **feasibility** dimension to decide if all the timing constraints can be met with a given set of $\{f_i\}\{a_i\}$ assignments. We will consider this problem in Section V. Then we will move on to the problem of **determining optimal $\{f_i\}\{a_i\}$ values** to minimize energy consumption while satisfying reliability and timing constraints at the same time (Section VI).

Notice that if the input to a specific problem instance is only the overall system target reliability Φ_{sys}^t, then the task-level reliability objectives $\{\Phi_i^t\}$ should be derived as the first step. In this case, a reasonable approach is to perform *uniform reliability scaling*, in such a way that the ratio $\frac{1-\Phi_i^t}{1-\Phi_i^0}$ is set to a common value Q across all tasks $T_1, \ldots, T_n$. Intuitively, this makes sure that the *PoF* improvement (or, degradation) uniformly applies to all the tasks, compared to the original *PoF* levels $\{1 - \Phi_i^0, \ i = 1, \ldots, n\}$.

V. Feasibility Conditions

Before addressing the problem of choosing optimal recovery allowance and frequency assignments, we need to consider the feasibility problem. Specifically, if a certain set of frequency $\{f_i\}$ and recovery $\{a_i\}$ assignments are considered for energy and reliability objectives, we must make sure that the job deadlines will be indeed met in *all* execution scenarios within the hyperperiod, where up to a_i instances of task T_i can incur transient faults and the corresponding dynamic recoveries are executed, for $i = 1, ..., n$.

Given a set of integers $z_1, \ldots, z_n$, a *fault pattern* [2] corresponds to the set of execution scenarios where exactly z_i distinct instances of T_i encounter transient faults during the hyperperiod H. If $z_i \leq a_i$ $(\forall i \ 1 \leq i \leq n)$, we call this pattern an $\{a_i\}$-*fault pattern*. For instance, assume $a_1 = 2, a_2 = 3$ and $a_3 = 1$ for three tasks T_1, T_2 and T_3. Any execution scenario where at most 2, 3 and 1 instances of T_1, T_2 and T_3 fail, respectively, constitutes a different fault

pattern with respect to $\{a_1 = 2, a_2 = 3, a_3 = 1\}$ allowance. If there are 3 instances of T_1 within the hyperperiod we need to consider all single and two-instance combinations for potential fault occurrences.

Obviously, the feasibility must be preserved for all distinct $\{a_i\}$-fault patterns for a given recovery allowance set $\{a_i\}$. At first, from a computational point of view, this appears as a prohibitively expensive task, because there are $\prod_{i=1}^{n} \sum_{j=1}^{a_i} \binom{k_i}{j}$ distinct $\{a_i\}$-fault patterns for a given recovery allowance set $\{a_i\}$, suggesting a potentially exponential number of cases to analyze.

Let $Z_{\{a_1,\ldots,a_n\}}$ be the set of all $\{a_i\}$-fault patterns corresponding to a given recovery allowance assignment $\{a_1, \ldots, a_n\}$. The following theorem, whose proof is provided in [29] due to space limitations, underlines that assessing feasibility under a single (and well-defined) worst-case fault occurrence pattern is necessary and sufficient.

Theorem V.1. *For a fixed frequency and recovery allowance assignment $\{a_1, \ldots, a_n\}$, the periodic task set remains feasible in every fault pattern $\in Z_{\{a_1,\ldots,a_n\}}$, if and only if all the deadlines can be met when the first a_i instances of every task T_i $(i = 1 \ldots n)$, encounter transient faults during the same execution.*

Moreover, the last condition can be checked by resorting to fault-sensitive processor-demand analysis techniques [2] as detailed in our technical report. The technique essentially consists in creating this *worst-case* fault scenario (where the first a_i instances of each task T_i need separate recoveries) and making sure that the *fault-sensitive* processor demand in every interval $(0, D)$ ($D \leq H$), does not exceed the length of the interval and has the complexity $O((\frac{H}{p_{min}}) \cdot n)$ where p_{min} is the smallest period among all tasks. It is sufficient to check the validity of this condition only at task period boundaries.

It is worthwhile to note that this result is along the lines of existing real-time scheduling theory which suggests that, when the workload of instances of given periodic task may change (for example, by intentionally *skipping* certain instances), the worst-case occurs with the so-called *deeply red* pattern – a scenario where all tasks present their maximum demand as early as possible and delay the *skips* as much as possible [7], [17], [22]. Our result essentially extends those results to fault-sensitive settings and has the same pseudo-polynomial time complexity.

VI. Recovery Allowance and Frequency Assignments

A. Static Phase

In this section, we address our main design problem: to determine frequency and recovery allowance assignments to minimize the total energy, while meeting task-level reliability and time constraints. In general, the non-trivial interplay of *reliability, timing* and *energy* dimensions makes the problem rather challenging. A given task's energy consumption is, in general, likely to decrease with decreasing frequency; but this may also violate the task's reliability constraint. Moreover, reducing a task's frequency and/or increasing its recovery allowance may also affect slow-down opportunities for other tasks.

Note that task-level target reliability objectives impose hard constraints on the frequency and recovery allowance assignment. In particular, if task T_i runs at discrete speed level $f_i = s_j$ $(1 \leq j \leq \ell)$, then Equation (4) suggests *the existence of a minimum recovery allowance value* a_i, to guarantee the reliability objective Φ_i^t. Consequently, we can construct an $n \times \ell$ table (*Minimum Recovery Table (MRT)*), where $MRT_{i,j} = min\{a_i | \Phi_i(f_i = s_j, a_i, k_i) \geq \Phi_i^t\}$. Intuitively the entry (i, j) of the table gives the minimum number of dynamic recoveries that must be assigned to task T_i when it runs at speed s_j to achieve Φ_i^t. It is clear that if $f_i = s_j$, the number of recoveries should be exactly $MRT_{i,j}$: a smaller value would violate the reliability constraint and a larger one would unnecessarily narrow the feasibility space of other tasks (see Theorem V.1).

Based on these observations, we propose two schemes. The first, *Lock-Step Frequency Scaling Algorithm (LSF)*, initially sets all task frequencies to $f_i = s_{max} = s_\ell$ and $a_i = MRT_{i,\ell}$. Then it iteratively attempts to reduce the frequency level of each task by one level, as long as the reliability and feasibility constraints are satisfied. Specifically, in a given iteration of *LSF*, a specific quantity $\delta_{i,j}$ is evaluated for each task. When the current frequency of task T_i is reduced from $f_i = s_{j+1}$ to s_j, we have:

$$\delta_{i,j} = \frac{k_i(E_i(s_{j+1}) - E_i(s_j))}{R_i(s_{j+1})^{k_i} - R_i(s_j)^{k_i}}$$

above $k_i(E_i(s_{j+1}) - E_i(s_j))$ indicates the energy savings obtained by total k_i job instances of T_i within the hyperperiod, when the task f_i is scaled down from s_{j+1} to s_j. Similarly, $R_i(s_{j+1})^{k_i} - R_i(s_j)^{k_i}$ is the reliability degradation that accompanies the same tentative speed reduction. Informally, $\delta_{i,j}$ is a measure of *utility* (i.e. energy savings per unit reliability degradation) corresponding to one-level speed scaling, guiding the algorithm's operation.

In a given iteration of *LFS*, a task T_i is said to be *eligible* for frequency scaling from s_{j+1} to s_j, if the task set remains feasible with $f_i = s_j$ and $a_i = MRT_{i,j}$ assignments (assuming other tasks' assignments remain the same). Let G_h be the set of *eligible* tasks in iteration h. *LFS* selects the task $T_i \in G_h$ for scaling down by one level as the one with maximum $\delta_{i,j}$ value; and stops when either G_h is empty or the energy-efficient frequency values for all tasks have been reached. As a result, *LFS* has at most $(\ell - 1)n$ iterations and each iteration performs at most n^2 feasibility checks. The complexity of *LFS* is therefore $O(\ell \cdot n^3(\frac{H}{P_{min}}))$.

We also experimented with a faster algorithm called

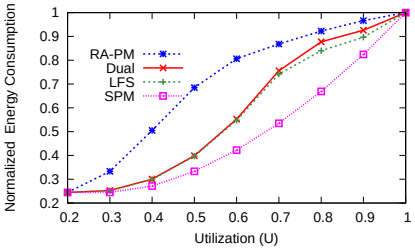


Figure 5. Impact of the utilization on the energy consumption with 10 tasks per set

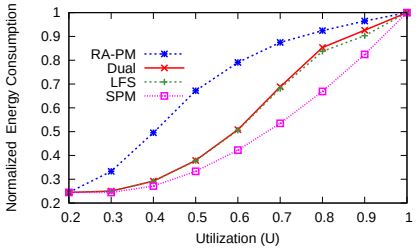


Figure 6. Impact of the utilization on the energy consumption with 20 tasks per set

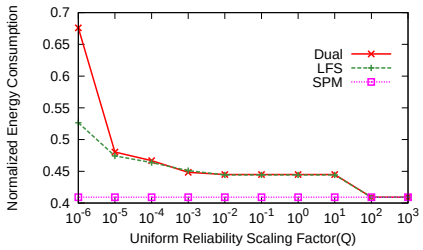


Figure 7. Impact of uniform reliability scaling factor on system energy consumption

the *Dual Speed (Dual)* Algorithm, that avoids checking the potential utility of scaling down every task, for every frequency level. Specifically, that algorithm first identifies the lowest *common* speed s_c for all tasks such that the feasibility is preserved with $\{f_i = s_c\}$ and $\{a_i = MRT_{i,c}\}$ assignments. These can be done in at most $log(\ell)$ steps by performing a binary search on the number of available ℓ discrete frequencies. Then it attempts to reduce the speed of some tasks to s_{c-1} while preserving feasibility, essentially using at most two distinct speed levels for the entire task set. The algorithm is based on the intuition that a uniform speed assignment is typically beneficial for real-time task sets, due to the convexity of dynamic power consumption. The dual-speed algorithm has the complexity $O((\log \ell + n) \cdot n(\frac{H}{P_{min}}))$. The pseudo-codes of both *LFS* and *Dual* algorithms can be found in our technical report [29].

Performance Evaluations: A discrete-event simulator was implemented in C programming language to evaluate the performance of our new schemes. In our simulations, in addition to *LFS* and *Dual* schemes, we implemented two additional schemes:

- The periodic *RA-PM* scheme [31], whose objective is to preserve the system's original reliability. We implemented the *largest-utilization-first (LUF)* and *smallest-utilization-first (SUF)* variations to determine the managed tasks [31]. The results were very similar, and we include only the results for *LUF* below.
- The *Static Power Management (SPM)* scheme [3], which computes optimal slow-down factors to minimize energy without considering reliability objectives. It is included in our comparison to assess the potential energy cost of provisioning for reliability.

A cubic frequency-dependent power component P_d is assumed and it is set to unity at the maximum processor frequency. The frequency-independent power component P_{ind} for each task is normalized with respect to P_d and we set $P_{ind} = 0.05$ in our simulations. The ten discrete frequency levels that we assume are modeled after Intel Xscale processor [24]. Also, Poisson distribution is used to simulate transient faults with an average fault rate of $\lambda_0 = 10^{-6}$ at the maximum frequency, which is a realistic fault rate as reported in [34]. In our simulations, we set the fault rate exponent d to 3.

Each point in the presented figures is obtained by averaging the results obtained through 1000 different task sets. The periods (p_i) of each task set are uniformly generated and fall in the range of $[10ms, 1080ms]$. The total utilization (U) of per task set is varied from 0.2 to 1.0 (full load) in steps of 0.1. The *UUniFast* algorithm [5] is used to generate the individual task utilizations (u_i). All energy consumption results are normalized with respect to the *no power management (NPM)* scheme that executes all tasks without any frequency and voltage scaling (i.e. at s_{max}).

First, we evaluate the impact of the utilization on the energy consumptions by setting $\Phi_i^t = \Phi_i^0$ (original reliability) for all tasks. Figure 5 shows the evaluation results with 10 tasks per task set. As expected, the energy consumption of all schemes increases with increasing U, due to the need for higher frequencies to preserve feasibility. The performance of *SPM* corresponds to maximum energy savings that can be obtained, *even when ignoring reliability objectives*. Compared to *RA-PM*, *Dual* and *LFS* have significant energy savings up to 50%. This is due to the dynamic recovery allocation strategy that can achieve reliability objectives with small number of recoveries, while leaving more slack for slowdown. In fact, when $U \leq 0.5$, the performance of our schemes approaches to that of SPM, because the use of small number of recoveries can be compensated by small increases in frequency levels in that region. It is also worthwhile to note that *LFS*, despite its more complex search algorithm, has only marginal gains over *Dual* and only at high utilization values, when the objective is to maintain original reliabilities. Figure 6 presents the same analysis this time for 20-task sets and we observe that the trends remain very similar. It is interesting to note that *Dual* approaches further *LFS* with increasing number of tasks as the algorithm has more chances to identify tasks for scaling down to s_{c-1} without violating feasibility.

Figure 7 shows the impact of the task-level target reliabilities (Φ_i^t) on the energy consumptions when $U = 0.5$ for 20-task sets. Here, the x axis is the *uniform reliability scaling factor* Q, defined in Section IV. Specifically, a value of $Q = 1$ corresponds to targeting original task-level reliabilities; a smaller (larger) scaling factor Q corresponds

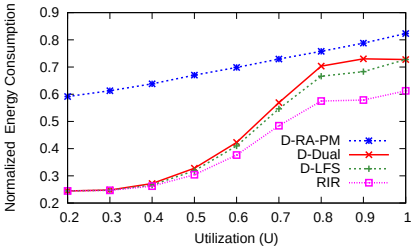


Figure 8. Impact of utilization PoF on system energy consumption

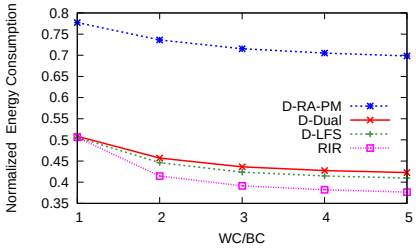


Figure 9. Impact of actual workloads on system energy consumption

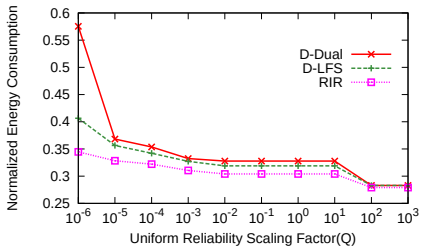


Figure 10. Impact of uniform reliability scaling factor on system energy consumption

to targeting smaller (larger) *PoF* figures. The *SPM* algorithm is reliability-ignorant, but its energy consumption figure is included as a comparison. Several observations are in order: when $Q \geq 10^2$, our algorithms' performance converges to that of *SPM* because they also choose to execute tasks without (or with very small number of) recoveries since the target reliability requirements are loose. However, with decreasing Q, the algorithms are forced to schedule additional recoveries and this causes some modest increase in energy consumption. It is interesting to note that in the region $[10^{-4}, 10^{-1}]$ the energy consumption of the schemes increases only marginally. This is due to the fact that the recoveries can be assigned only in integer units; and once a new recovery is added to the application, typically overall reliability improves by great margins. This is further enhanced by the existence of discrete speed levels; making *very fine-grained* reliability control somewhat difficult. However, for very high reliability requirements (e.g. $Q \leq 10^{-5}$) the algorithms are forced to provision for additional recoveries with a corresponding energy cost. In fact, one can notice that the comprehensive search mechanism of *LFS* starts to pay off when the required *PoF* levels are very low.

B. Dynamic extensions

The static algorithms *LFS* and *Dual* are designed to meet the problem's timing and reliability constraints under worst-case workload assumptions. However, it is well-known that, often the tasks' actual workloads during execution deviate from the worst-case, and significant amount of dynamic slack can be expected. In fact, exploiting the dynamic slack by reducing the processor speed as appropriate is a common strategy in DVFS frameworks [4], [19].

While similar opportunities exist in our settings, the dynamic reclaiming becomes more challenging due to hard reliability constraints. Specifically, reclaiming slack at run-time by dynamically reducing the speed, even if it guarantees the timing constraints, *can violate the task's given target reliability objectives*.

Therefore, we present a conservative but safe technique to perform dynamic slack reclaiming at runtime while still preserving the task-level reliability guarantees. Specifically, we use the DRA algorithm [4] to keep track of the dynamic slack, and evaluate the *earliness* of each task at dispatch time. However, a given job's speed is actually reduced only when the amount of safely reclaimable slack (earliness) is large enough to schedule also a new dynamically scheduled recovery – this limits the extent of achievable dynamic slow-down, but preserves the reliability constraint. Yet, when a job completes successfully (i.e. without a fault), the algorithm can re-use the slack of the new recovery to safely slow-down another job. Based on these principles, we extended *LFS* and *Dual Speed* schemes to dynamic settings, obtaining *D-LFS* and *D-Dual* algorithms, respectively.

Performance Evaluation: We implemented *D-LFS* and *D-Dual* algorithms in our simulator. We also implemented the *Dynamic RA-PM (D-RA-PM)* scheme, which uses the *wrapper task technique* [31], to perform slack reclaiming at runtime. Finally, we implemented a *Reliability-Ignorant-Reclaiming scheme (RIR)* which uses the entire dynamic slack for slow-down at run-time, without considering the potential reliability degradation. *RIR* is essentially used as a yardstick algorithm to assess the energy savings of our solutions.

We use the same evaluation settings as in the evaluation part of static algorithms. To model the variations in the actual workload, we use $\frac{WC}{BC}$ (varying from 1 to 5), which represents the ratio of worst-case execution time to the best- case execution time. The higher this ratio, the more the actual workload deviates from the worst case. We randomly generate the actual execution time of each job, by using normal distribution with the mean $\frac{WC+BC}{2}$ and standard derivation $\frac{WC-BC}{12}$ [4]. This guarantees that 99.7% of generated execution times fall in the range $[BC, WC]$ and values outside this range are not considered. For each points in the figures 1000 task sets (each with 20 tasks) are used; each execution is repeated 1000 times during the hyperperiod. All energy consumption results are again normalized with respect to NPM.

Figure 8 illustrates the energy consumptions as a function of total utilization (U), with $\frac{WC}{BC} = 5$ and $\Phi_i^t = \Phi_i^0$ for all tasks. The relative order of schemes remains the same with respect to Figure 5. However, now *D-RA-PM*'s performance relatively deteriorates compared to the static version. This is because *D-RA-PM* is based on the *wrapper-task* technique

which uses the maximum frequency as the nominal speeds of all jobs to preserve reliability when dynamically reducing the speed. However, our schemes use nominal frequency assignments from the static algorithms, which are relatively lower. Also we notice that our dynamic schemes no longer converge to *RIR* at high U values due to the need for reserving additional recovery to maintain reliability guarantees – *RIR*, by ignoring reliability, can reclaim full dynamic slack. Another observation is that with increasing utilizations, the difference of energy performances between *D-LFS* and *D-Dual* becomes more emphasized since the latter is restricted to the use of two speed levels during dynamic reclaiming.

Figure 9 shows the impact of variability in the actual workload (i.e. the $\frac{WC}{BC}$) on the energy performance, when $U = 0.6$ and $\Phi_i^t = \Phi_i^0$. In general, we find that the energy consumption decreases with increasing dynamic slack (higher $\frac{WC}{BC}$ ratio). The *D-Dual*'s limitations due to the use of two speeds restrict its ability to exploit fully the increasing dynamic slack. Finally, the hard requirement for maintaining the reliability objectives results in reduced ability for reclaiming slack, compared to *RIR*.

In Figure 10, we show the relationship between the task-level target reliabilities (Φ_i^t) and the energy consumptions, for $U = 0.5$ and $\frac{WC}{BC} = 5$. Again, we observe patterns similar to those observed in the case of static schemes (Figure 7). A notable difference is that the shortcomings of *D-Dual* are slightly more emphasized in dynamic settings.

VII. Closely Related Work

In recent years, DVFS has been investigated extensively to manage energy in real-time systems [4], [19]. However, the joint consideration of energy management and fault tolerance has attracted attention only very recently. By exploiting the primary/backup model, Unsal *et al.* proposed an energy-aware software-based fault tolerance scheme, where the execution of backup tasks is postponed as much as possible to minimize the overlap between primary and backup executions and thus to reduce energy consumption [23]. An adaptive checkpointing scheme to tolerate a fixed number of transient faults while minimizing energy consumption has been studied in [26]. Considering both re-execution and replication for fault tolerance, Izosimov *et al.* studied an optimization problem for mapping a set of tasks with reliability constraints, timing constraints and precedence relations to processors and determining appropriate fault tolerance policies for tasks [15]. Note that, these studies either focused on tolerating a fixed number of faults [15] or assumed a constant arrival rate for transient faults [27]. However, the negative effect of DVFS on system reliability due to increased number of transient faults at lower supply voltages (especially for the ones induced by cosmic ray radiations) has been shown [32]. Such negative effects have been confirmed in other studies as well [8], [12]. Ejlali *et al.* studied a number of schemes that combine the information about hardware resources and temporal redundancy to save energy and to preserve system reliability [11]. Pop *et al.* studied the energy and reliability trade-offs for distributed heterogeneous embedded systems [20]. More recently, Ejlali *et al.* studied a standby-sparing hardware redundancy technique for fault tolerance, where the standby processor is operated at low power state whenever possible provided that it can catch up and finish the tasks in time [10]. In our previous work, we proposed a reliability-aware power management (RA-PM) framework to preserve system reliability while exploiting slack time for energy savings [30], and extended it to periodic real-time tasks [31]. However, all these papers focused on preserving the original reliability of tasks without considering the different reliability requirements of individual tasks, which is the topic of this paper. More recently, we proposed the *Generalized Shared Recovery (GSHR)* technique [28], where a small number of recovery tasks are shared among tasks while satisfying an arbitrary *system-level reliability target*. In addition, *GSHR* considers only frame-based task systems, where all tasks share a common deadline. Also, in [33], we applied the weakly-hard real-time model to RA-PM settings. However, that technique is an individual-recovery based scheme where jobs without statically scheduled recoveries have to be still executed at the maximum frequency. In this work, we differentiate the reliability requirements of individual periodic tasks and study recovery allowance based schemes, where all jobs can be scaled down and recoveries are dynamically allocated.

VIII. Conclusions

In this research effort, we presented a framework to provide arbitrary task-level reliability guarantees to periodic real-time tasks, while controlling energy consumption through DVFS. As opposed to the existing reliability-aware power management (RA-PM) frameworks, where the aim is to preserve the original reliability and the recovery jobs are statically allocated to scaled jobs, we introduced the concept of *recovery allowances* that can be reclaimed anywhere during the hyperperiod. This flexibility helps to improve reliability significantly, with minimum in-advance reservation for potential recoveries. We presented static and dynamic algorithms that are shown to reduce energy consumption while maintaining the feasibility and reliability objectives. To the best of our knowledge, this is the first work to consider task-level reliability objectives in periodic execution settings. An interesting future work direction is to consider multi-processor settings where both permanent and transient faults could be potentially tolerated through a mix of hardware and time redundancy techniques.

Acknowledgments

This work was supported by US National Science Foundation awards CNS-1016855, CNS-1016974, and CAREER Awards CNS-0546244 and CNS-0953005.

REFERENCES

[1] P. M. Alvarez, H. Aydin, R. Melhem, and D. Mosse. Scheduling optional computations in fault-tolerant real-time systems. In *Proc. International Conf. on Real-Time Computing Systems and Applications (RTCSA)*, 2000.

[2] H. Aydin. Exact fault-sensitive feasibility analysis of real-time tasks. *IEEE Transactions on Computers*, 56(10):1372–1386, 2007.

[3] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. In *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, pages 313–322, Dec. 2006.

[4] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. on Computers*, 53(5):584–600, 2004.

[5] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2), 2005.

[6] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proc. of The HICSS Conference*, Jan. 1995.

[7] M. Caccamo and G. C. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 1997.

[8] V. Degalahal, L. Li, V. Narayanan, M. Kandemir, and M. J. Irwin. Soft errors issues in low-power caches. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 13(10):1157–1166, Oct. 2005.

[9] V. Devadas and H. Aydin. On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications. In *Proc. ACM Conference on Embedded Systems Software (EMSOFT'08)*, 2008.

[10] A. Ejlali, B. M. Al-Hashimi, and P. Eles. A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems. In *Proc. of the Int'l conference on Hardware/software codesign and system synthesis (CODES)*, 2009.

[11] A. Ejlali, M. T. Schmitz, B. M. Al-Hashimi, S. G. Miremadi, and P. Rosinger. Energy efficient seu-tolerance in dvs-enabled real-time systems through information redundancy. In *Proc. of the Int'l Symposium on Low Power and Electronics and Design (ISLPED)*, 2005.

[12] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner. Razor: circuit-level correction of timing errors for low-power operation. *IEEE Micro*, 24(6):10–20, 2004.

[13] P. Hazucha and C. Svensson. Impact of cmos technology scaling on the atmospheric neutron soft error rate. *IEEE Trans. on Nuclear Science*, 47(6):2586–2594, 2000.

[14] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and modeling of computer reliability as affected by system activity. *ACM Trans. Comput. Syst.*, 4(3):214–237, 1986.

[15] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *Proc. of the conference on Design, Automation and Test in Europe (DATE)*, 2005.

[16] R. Jejurikar and R. Gupta. Dynamic voltage scaling for system wide energy minimization in real-time embedded systems. In *Proc. of the Int'l Symposium on Low Power Electronics and Design (ISLPED)*, pages 78–81, 2004.

[17] G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 1995.

[18] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20(1):46–61, Janary 1973.

[19] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for lowpower embedded operating systems. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[20] P. Pop, K. Poulsen, V. Izosimov, and P. Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *Proc. of the Int'l Conference on Hardware/software codesign and System Synthesis (CODES+ISSS)*, pages 233–238, 2007.

[21] D. K. Pradhan. *Fault Tolerance Computing: Theory and Techniques*. Prentice Hall, 1986.

[22] G. Quan and X. Hu. Enhanced fixed-priority scheduling with (m,k)- firm guarantee. In *Proc. IEEE Real-Time Systems Symposium (RTSS)*, 2000.

[23] O. S. Unsal, I. Koren, and C. M. Krishna. Towards energy-aware software-based fault tolerance in real-time systems. In *Proc. of The International Symposium on Low Power Electronics Design (ISLPED)*, Aug. 2002.

[24] R. Xu, D. Mossé, and R. Melhem. Minimizing expected energy consumption in real-time systems through dynamic voltage scaling. *ACM Trans. for Embedded Computing Systems*, 25(4), 2007.

[25] Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *Proc. of the conference on Design, Automation and Test in Europe (DATE)*, 2003.

[26] Y. Zhang and K. Chakrabarty. Task feasibility analysis and dynamic voltage scaling in fault-tolerant real-time embedded systems. In *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference(DATE)*, 2004.

[27] Y. Zhang, K. Chakrabarty, and V. Swaminathan. Energy-aware fault tolerance in fixed-priority real-time embedded systems. In *Proc. of the 2003 IEEE/ACM int'l conference on Computer-aided design*, 2003.

[28] B. Zhao, H. Aydin, and D. Zhu. Generalized reliability-oriented energy management for real-time embedded applications. In *Proc. the Design Automation Conference (DAC)*, 2011.

[29] B. Zhao, H. Aydin, and D. Zhu. Energy management under general reliability constraints: the periodic case. In *Technical Report - GMU Computer Science Dept.*, 2011. Available at http://cs.gmu.edu/~aydin/tr-2011-29.pdf.

[30] D. Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2006.

[31] D. Zhu and H. Aydin. Reliability-aware energy management for periodic real-time tasks. *IEEE Trans. on Computers*, 58(10):1382–1397, 2009.

[32] D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *Proc. of the Int'l Conf. on Computer Aidded Design*, 2004.

[33] D. Zhu, X. Qi, and H. Aydin. Energy management for periodic real-time tasks with variable assurance requirements. In *Proc. of the IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[34] J. F. Ziegler. Trends in electronic reliability: Effects of terrestrial cosmic rays. available at http://www.srim.org/SER/SERTrends.htm, 2004.