Improving Reliability for Real-Time Systems through Dynamic Recovery

Yue Ma[1], Thidapat Chantem[2], Robert P. Dick[3], and X. Sharon Hu[1]

[1]Department of CSE, University of Notre Dame, Notre Dame, IN 46656, USA, {yma1,shu}@nd.edu
[2]Department of ECE, Virginia Polytechnic Institute and State University, Arlington, VA, 22203, tchantem@vt.edu
[3]Department of EECS, University of Michigan, Ann Arbor, MI 48109, dickrp@umich.edu

***Abstract*—Technology scaling has increased concerns about transient faults due to soft errors and permanent faults due to lifetime wear processes. Although researchers have investigated related problems, they have either considered only one of the two reliability concerns or presented simple recovery allocation algorithms that cannot effectively use available time slack to improve soft-error reliability. This paper introduces a framework for improving soft-error reliability while satisfying lifetime reliability and real-time constraints. We present a dynamic recovery allocation technique that guarantees to recover *any* failed task if the remaining slack is adequate. Based on this technique, we propose two scheduling algorithms for task sets with different characteristics to improve system-level soft-error reliability. Lifetime reliability requirements are satisfied by reducing core frequencies for appropriate tasks, thereby reducing wear due to temperature and thermal cycling. Simulation results show that the proposed framework reduces the probability of failure by at least 8% and 73% on average compared to existing approaches.**

***Keywords*—Soft-error reliability; Lifetime reliability; Dynamic recovery; Real-time embedded system.**

I. Introduction

Since many real-time embedded systems are used in safety critical applications and may be expensive as well as difficult to replace, soft-error reliability (SER) due to transient faults as well as lifetime reliability (LTR) due to permanent faults are important design considerations. Techniques for reducing power, energy, and/or temperature through reducing core frequency to improve LTR, however, reduce SER, and vice versa. Motivated by applications such as infotainment systems [1], we focus on improving system-level SER while satisfying some pre-defined LTR and deadline requirements.

Although most existing work either targets SER or LTR [2]–[8], a few recent papers have examined both [9]–[11]. Das et al. aimed to maximize the minimum of SER and LTR by mapping tasks and increasing core frequencies [9]. Ma et al. proposed to maximize SER subject to LTR constraints on "big-little" architectures [11]. Both work improve SER while considering LTR for systems running real-time tasks, but neither leverages recovery techniques, which are effective methods to further improve SER. Zhou et al. proposed a technique to maximize system availability by allowing failed tasks to re-execute and increasing core frequencies to improve SER [10]. However, this greedy approach allocates recovery statically, which is less effective than dynamically allocate recoveries.

In this paper, we are interested in maximizing SER while satisfying real-time and LTR constraints. In order to maximize SER, we introduce a novel dynamic recovery technique where slack is shared and *all* failed tasks can be recovered if there is sufficient remaining slack for executing the tasks. A task is recovered by executing again from the beginning, and we assume all tasks are allowed to re-execute. We also assume faults are detected at the end of each task and that detection overhead can be ignored [12]. Compared to existing approaches that limit the number of recoveries [4] or assign redundancies to tasks statically [7], our technique provides the highest level of slack sharing flexibility, as will be shown in Section V. Since the effectiveness of the dynamic recovery technique depends on the task execution order, we explore how task scheduling can affect system-level SER and present two scheduling algorithms to improve the system-level SER for tasks with different characteristics.

We have developed a soft-error reliability improvement framework, referred to as `RIF`, to solve the problem identified above. `RIF` is composed of two components. The first one aims at increasing SER through task scheduling (i.e., determining the task priorities) and dynamic recovery. If LTR is lower than a predefined limit, the second component of `RIF` is used to increase LTR subject to real-time constraints by reducing core frequencies for appropriate tasks. We make three main contributions. (i) We propose a new dynamic recovery allocation technique and derive the corresponding system-level SER. (ii) We introduce two scheduling algorithms for task sets with different characteristics to improve system-level SER. The first algorithm is computationally efficient and supports the special case where the execution times of tasks in a task set are similar and the amount of available slack is small. The second algorithm is more powerful and handles general task sets but at a higher time complexity. (iii) We devise a less costly method to obtain system-level SER if a task set belongs to the special case in (ii).

We evaluated `RIF` in a simulator, which is constructed based on the Nvidia Jetson TK1 board [13] with tasks from the MiBench benchmark suite [14]. `RIF` reduces the probability of failure by at least 8% and 73% on average when compared

This work was supported in part by NSF under awards CNS-1319904, CNS-1319718, and CNS-1319784.

to existing approaches.

II. System models and problem formulation

In this section, we first introduce the system models. We then outline the problem of interest and present an overview of our framework.

A. Task model

We consider a frame-based task set composed of independent tasks. Tasks are executed on a DVFS-enabled multicore processor where each core supports L frequency levels. The frequency levels are sorted in increasing order such that the L^{th} level, l_L, has the highest frequency. We assume that tasks are allocated to cores at design time (similar to partitioned scheduling [15]) and task migration between cores is not allowed [4], [11], [15], [16]. We also assume a task and its recovery should execute on the same core.

A task, τ_i, is associated with a tuple $\{f_i(l_i),\ c_i(l_i),\ p_i\}$ where $f_i(l_i)$ and $c_i(l_i)$ are the core frequency and worst-case execution time of τ_i if the core is running at frequency level l_i, respectively. p_i is $\tau_i's$ priority (a large value for p_i denotes a higher priority), and tasks' priorities determine their execution order. Since we aim to guarantee the real-time constraint for each task, any changes to priorities are acceptable as long as they do not violate timing requirements. Tasks should complete their executions by a common deadline, D. The slack is used to re-execute failed tasks and a core's slack, s, is defined as $D - \sum_{i=1}^{\Pi} c_i(l_i)$ where Π is the tasks executing on the core. Since tasks have the same period, we do not need to distinguish instances (i.e., jobs) of a task.

B. Soft-error and lifetime reliability

The soft-error reliability of a task is the probability that this task successfully completes. For task τ_i executing at the core frequency level l_i, $r_i^t(l_i)$ is the probability that no soft error occurs during its normal execution. $r_i^t(l_i)$ depends on the core frequency level and the execution time of τ_i [2], [9], [10],

$$r_i^t(l_i) = e^{-\lambda(l_i)\frac{c_i(l_i)}{f_i(l_i)}}, \tag{1}$$

where $\lambda(l_i)$ is the fault arrival rate and

$$\lambda(l_i) = \lambda_L 10^{\frac{d\times(1-f_i(l_i))}{1-f_i(l_1)}}, \tag{2}$$

where l_1 is the lowest core frequency level and λ_L is the average fault arrival rate when executing at the highest frequency level. d is a hardware specific constant that indicates the sensitivity of fault rates on frequency scaling.

Lifetime reliability depends on multiple wear-out effects. Wear due to electromigration, stress migration, and time-dependent dielectric breakdown is exponentially dependent on operating temperature. Wear due to thermal cycling depends on the amplitude (e.g., the difference between the peak and valley temperature), period, and cycle maximum temperature [17]. To improve LTR, both the operating temperature and the effects of thermal cycling should be reduced. We use a Monte Carlo simulation based modeling tool [17] to obtain system-level LTR and evaluate our framework, but our work is independent on the underlying reliability modeling tool.

C. Problem formulation

Given the importance of SER, LTR, and real-time requirements, we aim to solve the problem of maximizing system-level SER for each core, R_{sys}, while satisfying real-time and LTR (measured by the mean-time-to-failure (MTTF) [5], [17]) constraints:

$$\sum_{\tau_i\in\Pi} c_i(l_i) \ \leq \ D, \tag{3}$$

$$MTTF \ \geq \ MTTF_{TH}, \tag{4}$$

where $MTTF_{TH}$ is the minimum MTTF that the system must achieve [16] and Π is the tasks executing on the core. R_{sys} is defined as the probability that all tasks complete successfully. A task is considered to be successful if it encounters no soft error during its first execution or successfully completes in the second execution (where the second execution is the recovery). Hence, R_{sys} is not only determined by the SER of each task, but also the recovery technique. We will discuss how to obtain R_{sys} with our proposed dynamic recovery technique in Section III-A. Note that we assume task migration between cores is not allowed and tasks execute on the same core with their recoveries. Since the solutions are independent on different cores, we propose to solve the problem for one core, and the solution can be applied to other cores.

D. Overview of framework

We propose a soft-error reliability improvement framework (`RIF`) to solve the problem defined above. `RIF` consists of two components. One focuses on improving system-level SER and the other on satisfying LTR and real-time constraints. In order to maximize system-level SER, the first component allows tasks to run at the highest core frequency and improves system-level SER through dynamic recovery and task scheduling. We first derive how to calculate system-level SER with our dynamic recovery technique and discuss how task scheduling affects the system-level SER. Then, we describe an efficient scheduling algorithm for task sets with some special properties and a more powerful scheduling algorithm for general task sets. The second component of `RIF` checks whether the LTR constraint (in (4)) is satisfied. If not, core frequencies of lower priority and power-hungry tasks are iteratively reduced until both the LTR and real-time constraints are satisfied. In the subsequent sections, we will discuss the details.

III. Improving soft-error reliability

In this section, we first introduce our dynamic recovery technique and discuss how task scheduling affects system-level SER. We allow all tasks to run at the highest core frequency, and then schedule tasks to improve the system-level SER.

A. Dynamic recovery technique

We propose a new approach to dynamically allocate recoveries to failed tasks. In our technique, available slack is shared by *all* tasks and dynamically assigned on a first-come, first-serve basis. A recovery is allocated to task τ_i if the remaining slack is no smaller than $c_i(l_L)$. The recoveries are required to execute at the highest core frequency. Although a recovery may

still fail due to the occurrence of soft errors, the probability that both a task and its recovery fail is very low. Therefore, we only allocate one recovery for each task to prevent a task from consuming too much slack[1]. Since a recovery is allocated only when the remaining slack is adequate, the recovery itself will not violate the real-time requirement. At the same time, whether a high priority task consumes slack affects recovery allocation of a lower priority task since the higher priority task is executed first. Hence, task scheduling directly impacts system-level SER.

We now discuss how to obtain the system-level SER with a given schedule $\mathcal{S}$. For a schedule, $r_i^s(\mathcal{S})$ denotes the probability that τ_i has a recovery and the recovery completes successfully. Hence, the probability that a task completes successfully is

$$r_i(\mathcal{S}) = 1 - (1 - r_i^t) \times (1 - r_i^s(\mathcal{S})). \tag{5}$$

It follows that the system-level SER is

$$R_{sys}(\mathcal{S}) = \prod_{i=1}^{n} \{1 - (1 - r_i^t)(1 - r_i^s(\mathcal{S}))\}. \tag{6}$$

$R_{sys}(\mathcal{S})$ denotes the system-level SER is determined by $\mathcal{S}$.

The variable $r_i^s(\mathcal{S})$ is the key to obtain the $R_{sys}(\mathcal{S})$. We use a concept called execution pattern to calculate $r_i^s(\mathcal{S})$. Suppose $\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$ where task τ_i has a higher priority than τ_{i+1}. We use the execution pattern, $\mathcal{P}_i$, to indicate the successful and failed tasks before τ_i. Inside an execution pattern, τ_k^+ denotes the successful completion of τ_k and τ_k^- indicates that τ_k has failed. Since one execution pattern is composed of $i-1$ tasks, there are 2^{i-1} patterns for τ_i, and we use $\mathcal{P}_{i,j}$ to indicate the j^{th} pattern. The value of $r_i^s(\mathcal{S})$ can be determined if we know the time used by the recovered tasks in each pattern, denoted by $T(\mathcal{P}_{i,j})$, and the probability that each pattern appears, denoted by $Prob(\mathcal{P}_{i,j})$. Suppose there are m patterns satisfying

$$T(\mathcal{P}_{i,j}) + c_i(l_L) \leq s, \tag{7}$$

where s is the shared slack. Then $r_i^s(\mathcal{S})$ can be calculated by

$$r_i^s(\mathcal{S}) = \begin{cases} r_i^t \times \sum_{j=1}^{m} Prob(\mathcal{P}_{i,j}), & \text{if } m > 0, \\ 0, & \text{otherwise,} \end{cases} \tag{8}$$

where $m = 0$ means that τ_i can never be recovered.

The key to obtain $r_i^s(\mathcal{S})$ is to calculate $T(\mathcal{P}_{i,j})$ and $Prob(\mathcal{P}_{i,j})$ for each pattern. $T(\mathcal{P}_{i,j})$ can be obtained as follows. A task τ_i uses slack to recover only when the remaining slack is larger than $c_i(l_L)$, so it is possible that a failed task with large execution time does not use the slack. We search each failed task in $\mathcal{P}_{i,j}$ and calculate $T(\mathcal{P}_{i,j})$ iteratively. We first initialize $T(\mathcal{P}_{i,j}) = 0$. Then, for each failed task τ_k in $\mathcal{P}_{i,j}$, if $c_k(l_L) \leq s$, update $T(\mathcal{P}_{i,j}) = T(\mathcal{P}_{i,j}) + c_k(l_L)$ and $s = s - c_k(l_L)$. For $Prob(\mathcal{P}_{i,j})$, note that it simply depends on the SER of each task included in the pattern. That is,

$$Prob(\mathcal{P}_{i,j}) = \prod_{\tau_k^+ \in \mathcal{P}_{i,j}} r_k^t \times \prod_{\tau_k^- \in \mathcal{P}_{i,j}} (1 - r_k^t). \tag{9}$$

Based on Eq. (6), task scheduling affects the system-level SER. We next introduce two scheduling algorithms for task sets with different characteristics to improve the system-level SER.

[1]The presented technique can be extended to allow some critical tasks to receive more than one recovery. The details are left to future work.

B. An efficient scheduling algorithm

We propose an efficient soft-error reliability improvement scheduling algorithm (ERIS) to improve system-level SER if the task sets satisfy certain conditions.

The algorithm is built on an observation that a schedule allowing more tasks to recover does not always leads to a higher system-level SER. For a given slack, allowing tasks with short execution time to recover may prevent tasks with long execution time to re-execute and finally reduce the overall system-level SER. Only when task sets satisfy the following conditions, allowing more tasks to recover leads to a higher system-level SER.

Theorem 1: If a core has n tasks, and the slack, s, satisfies the following conditions,

(i) $s < c_{\max}$,
(ii) $(n-1)c_{\min} \geq 2c_{\max}$,

then a schedule allowing more tasks to recover always leads to a higher system-level SER. $c_{\min}$ and $c_{\max}$ are shortest and longest execution time when the core frequency at the highest level, respectively. We omit the proof due to page limit.

Theorem 1 provides some intuition on how to schedule tasks. That is, the schedule that maximizes the number of tasks capable of being recovered would maximize the system-level SER. Observe that a task with a high priority and which executes earlier has a larger probability of recovering because less slack is consumed by prior tasks. In order to preserve slack for later tasks, high priority tasks should have short execution times. Hence, we propose an efficient scheduling algorithm (ERIS) that assigns the priority to tasks according to their execution times. Given a task set that satisfies the conditions in Theorem 1 and is scheduled according to ERIS, i.e., $p_i < p_j$ if $c_i(l_L) > c_j(l_L)$, then the system-level SER is maximized.

Since obtaining the system-level SER is useful not only for evaluating our proposed framework, but also in other work, e.g., reliability-aware energy management [4], we develop a method to obtain the system-level SER in pseudo-polynomial time for task sets that satisfy Theorem 1 and which are scheduled by ERIS. This method reduces the overhead in finding the execution patterns satisfying (7). We simplify $r_i^s(\mathcal{S})$ to r_i^s whenever doing so does not introduce ambiguity.

Suppose a schedule of n tasks is $\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$, where $p_i > p_{i+1}$ and $c_i(l_L) < c_{i+1}(l_L)$. For τ_i, we introduce two concepts: i) *heavy* set, Ω_i^+, is a subset of tasks in $\{\tau_1, \ldots, \tau_{i-1}\}$ where if all tasks in Ω_i^+ fail, the slack needed to recover all tasks in Ω_i^+ is larger than $s - c_i(l_L)$; ii) *light* set, Ω_i^-, is a subset of tasks in $\{\tau_1, \ldots, \tau_{i-1}\}$ where if all tasks in Ω_i^- fail, the slack needed to recover is smaller than or equal to $s - c_i(l_L)$. Tasks in both Ω_i^+ and Ω_i^- are sorted by decreasing priority. Let $\Omega_{i,j}^-$ be the j^{th} light set for task τ_i. If $\Omega_{i,j}^- = \{\tau_1, \tau_2\}$, it means that the combined slack needed by τ_1 and τ_2 to recover is smaller than or equal to $s - c_i(l_L)$. Based on this definition, one light set coincides with one execution pattern satisfying (7). Hence, finding all the light sets is equivalent to finding all the execution patterns satisfying (7), which can then be used to compute the system-level SER. We state three lemmas used to find light sets. The proofs are omitted due to page limit.

Lemma 1: With ERIS, a set Ω_i is a heavy set for τ_i only when $\sum_{\tau_j \in \Omega_i} c_j(l_L) > s - c_i(l_L)$.

Lemma 2: If $\Omega^+_{i,j}$ is a heavy set of τ_i, $\Omega^+_{i,k}$ is also a heavy set if $\Omega^+_{i,k}$ is a super set of $\Omega^+_{i,j}$, i.e., $\Omega^+_{i,j} \subset \Omega^+_{i,k}$.

Lemma 3: With ERIS, if $\Omega^+_{i,j}$ is a heavy set of τ_i, it is also a heavy set of τ_{i+1}.

For a task τ_i, we divide all its light sets into groups. $G_{i,k}$, the k^{th} group of τ_i's light sets, consists of light sets in which each set has exactly k tasks. Based on this definition, for task τ_i, there are i groups, i.e., $G_{i,0}$ to $G_{i,i-1}$. Based on Lemmas 1–3 and the definition of $G_{i,k}$, we provide a guideline on how to find light sets for each task, which will, in turn, help to maximize system-level SER.

Theorem 2: For task sets that satisfy Theorem 1 and which are scheduled by ERIS, if a light set, Ω^-, in $G_{i-1,k}$ satisfies $\sum_{\tau_j \in \Omega^-} c_j(l_L) \leq s - c_i(l_L)$, Ω^- is also a light set in $G_{i,k}$. If Ω^- is a light set in $G_{i-1,k-1}$, it is also a light set in $G_{i,k}$ if $c_{i-1}(l_L) + \sum_{\tau_j \in \Omega^-}(c_j(l_L)) \leq s - c_i(l_L)$.

We omit the proof of Theorem 2 due to page limit. Based on Theorem 2, we find all light sets for each task iteratively following a dynamic programming strategy.

The details of our method is shown in Alg. 1. We initialize $G_{i,0}$ for each task in Lines 2–8. $G_{i,0}$ is initialized to $\{\emptyset\}$ if $c_i(l_L) \leq s$, which means τ_i can be recovered if the slack is not consumed by other tasks. $G_{i,0} = \emptyset$ means τ_i can never be recovered. We construct $G_{i,k}$ for each task in Lines 9–28. Based on Theorem 2, $G_{i,k}$ is first set to $\emptyset$ (in Line 11) and then additional light sets are added to $G_{i,k}$ in Lines 12–22. For each light set in $G_{i-1,k}$, if it is still a light set for τ_i, it is added into $G_{i,k}$ (in Lines 12–16). Similarly, for each light set in $G_{i-1,k-1}$, we determine whether the set remains light if τ_{i-1} is added (in Lines 17–22). Based on Lemma 2, if $G_{i,k} = \emptyset$, groups from $G_{i,k+1}$ to $G_{i,i-1}$ are also equal to $\emptyset$ (in Lines 23–26). Finally we return groups of light sets $\{G_{i,0}, \ldots, G_{i,i-1}\}$ (in Line 29). The complexity of Alg. 1 is $O(n \times K)$ where $K(K \leq 2^n)$ is the number of returned light sets.

Algorithm 1 Find Light Sets

```
1: procedure FIND_SET(S = {τ1, τ2, ..., τn})
2:     for task τi in {τ1, τ2, ..., τn} do
3:         if ci(lL) ≤ s then
4:             Gi,0 = {∅}
5:         else
6:             Gi,0 = ∅
7:         end if
8:     end for
9:     for task τi in {τ2, τ3, ..., τn} do
10:        for k in {1, 2, ..., i − 1} do
11:            Gi,k = ∅
12:            for light set Ω− in Gi−1,k do
13:                if Σ_{τj∈Ω−}(cj(lL)) ≤ s − ci(lL) then
14:                    Gi,k = {Gi,k ∪ Ω−}
15:                end if
16:            end for
17:            for light set Ω− in Gi−1,k−1 do
18:                if ci−1(lL) + Σ_{τj∈Ω−}(cj(lL)) ≤ s − ci(lL) then
19:                    Ω− = {Ω− ∪ τi−1}
20:                    Gi,k = {Gi,k ∪ Ω−}
21:                end if
22:            end for
23:            if Gi,k = ∅ then
24:                Gi,k+1 = ··· = Gi,i−1 = ∅
25:                break
26:            end if
27:        end for
28:    end for
29:    return {Gi,0, ..., Gi,i−1} for i in {1, 2, ..., n}
30: end procedure
```

C. A general scheduling algorithm

Although ERIS is more efficient and effective in scheduling tasks satisfying the conditions in Theorem 1, for general task sets, it may lead to suboptimal schedule. In this subsection, we present a general soft-error reliability improvement scheduling algorithm (GRIS) for general task sets where a task's execution time can be any arbitrary value. GRIS guarantees the system-level SER is always higher than the static recovery techniques [4], [10]. GRIS first finds the optimal solution, set Φ, for the static recovery allocation problem. It then elevates the priority of tasks in Φ to further improve SER.

It can be seen that solving the static recovery allocation problem is the key to GRIS. This static recovery allocation problem is a variation of the knapsack problem, which can be solved using dynamic programming. Let $\Phi\{i, s'\}$ be a set of tasks that achieves the maximum system-level SER where some tasks in $\{\tau_1, \tau_2, \ldots, \tau_i\}$ can be recovered under the constraint that the slack is less than or equal to s'. Given this construction, a dynamic program for solving the knapsack problem can be leveraged to find the optimal solution, $\Phi\{n, s\}$. We omit the details due to space limit.

Based on the solution to the static allocation problem, tasks in Φ have a higher priority than tasks not in Φ. The priority is assigned in such a way that if $c_i(l_L) > c_j(l_L)$ then $p_i < p_j$ for both τ_i and τ_j in Φ. With dynamic recovery allocation, this scheduling algorithm guarantees that tasks in Φ can always be recovered. Tasks not in Φ can still have a recovery if some tasks in Φ successfully complete their execution. Hence, GRIS achieves a higher system-level SER than the optimal solution to the static recovery allocation problem. Since GRIS is a variation of knapsack problem, it can be completed in $O(n \times K)$ where K is the number of tasks in $\Phi\{n, s\}$. Compared to ERIS, although GRIS is more complicated, simulation results in Section V show that GRIS is more effective at improving system-level SER than ERIS in some cases.

IV. Satisfying lifetime reliability

To improve SER, it is desirable to execute tasks at the highest core frequency. However, this core frequency may violate the LTR constraint in (4). To address this problem, we propose to drop tasks' core frequencies to guarantee the LTR requirement. Dropping a task's core frequency reduces the power and operating temperature, but in turn may violate tasks' timing requirements and increase the arrival rate of transient faults (in Eq. (2)). In order to solve this trade-off problem, we propose a heuristic to reduce core frequencies for appropriate tasks. Note that although we reduce core frequencies of tasks,

recoveries always execute at the highest core frequency. Since the probability that a task fails is very low, running a recovery at the highest core frequency does not significantly impact long-term LTR.

We consider the LTR due to both operating temperature and thermal cycling. Reducing core frequencies for tasks is effective in reducing the operating temperature but may introduce thermal cycles. For the task model and dynamic recovery allocation technique under consideration, a core is only active in the earlier part of a period. Hence, if all tasks are running at the same frequency, there is only one thermal cycle in each period. However, if the frequency of task τ_i is lower or higher than both τ_{i-1} and τ_{i+1}, more thermal cycles may be introduced. Hence, we aim to reduce the core frequencies of tasks while avoiding additional thermal cycles.

Reducing a task's core frequency increases its execution time and reduces both its SER and available slack. This task, in the end, has a higher failure probability and is more likely to consume slack for recovery. Since a high-priority task first consumes slack to recover, and may affect the reliability of low-priority tasks, we adopt the general principle that high-priority tasks execute at the highest frequency while lowering the core frequency of low-priority tasks. We present a method to choose a task to reduce its core frequency in such a way that doing so maximizes the power saving and minimizes the influence on other tasks' reliability. We define $\Re_i(l_i = j, l_i = k)$ as the power-time ratio of τ_i when reducing core frequency from the j^{th} to the k^{th} level,

$$\Re_i(l_i = j, l_i = k) = \frac{\rho_i(l_i = j) - \rho_i(l_i = k)}{c_i(l_i = k) - c_i(l_i = j)}, \tag{10}$$

where $\rho_i(l_i = j)$ and $c_i(l_i = j)$ are power consumption and worst-case execution time, respectively, when τ_i running at the j^{th} frequency level. For a given schedule $\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$ where τ_1 has the highest priority and τ_n has the lowest one, we reduce the core frequencies of appropriate tasks (see Alg. 2). We iteratively reduce tasks' core frequencies until the MTTF is larger than a given threshold (in Lines 3–11). In each iteration, we select the task such that reducing one level of this task's core frequency achieves the largest $\Re$ and does not violate the deadline constraint. Meanwhile, in order to avoid introducing more thermal cycles, we only select the task that has a higher core frequency than low-priority tasks, i.e., $f_i > f_{i+1}$ (in Line 5). After reducing the selected task's (i.e., τ_t's) core frequency level, l_t, a new MTTF and slack s are calculated (in Line 10). This algorithm costs at most $n \times L$ iterations to find appropriate frequencies for tasks.

Algorithm 2 Frequency Reduction

```
1:  procedure FREQ_RED(S)
2:      ℜ_max = 0, τ_t = τ_n
3:      while MTTF_sys < MTTF_TH do
4:          for τ_i ∈ {τ_1, ..., τ_{n-1}} do
5:              if s > c_i(l_i − 1) − c_i(l_i) and ℜ_i(l_i, l_i − 1) > ℜ_max
    and f_i > f_{i+1} then
6:                  ℜ_max = ℜ_i(l_i, l_i − 1)
7:                  τ_t = τ_i
8:              end if
9:          end for
10:         f_t(l_t) = f_t(l_t − 1) and update s and MTTF_sys
11:     end while
12: end procedure
```

V. Evaluation

We evaluate the proposed RIF by conducting simulations and comparing it with existing approaches.

A. Simulation setup

We compared RIF with two representative approaches: generalized shared recovery approach (GSR) [4] and partial replication and speedup approach (PRS) [10]. GSR is composed of two modules; one allocates recovery and the other reduces core frequency to minimize power consumption. We kept the first GSR module but replaced the second module with Alg. 2 to satisfy the LTR constraint. PRS is a greedy algorithm based approach to maximize the minimum of LTR and SER by allocating recoveries offline. We evaluated RIF with ERIS in Section III-B (RIF-ERIS) and with GRIS in Section III-C (RIF-GRIS). The probability of failure (PoF), which is defined as $1 - R_{sys}$, is used as a metric for comparison. We also set $\lambda_L = 10^{-6}$ and $d = 3$ (in Eq. (2)) [10]. Note that λ_L and d may have different values for different hardware platforms, but the improvement of RIF is independent of their values.

The comparisons were conducted on a simulator, which is constructed based on a Nvidia's Jetson TK1 board [13]. The operating temperature is calculated using an resistance-capacitance thermal modeling tool and thermal parameters are extracted from the TK1 board [16]. We specified two different configurations for the task set. In the first configuration, a task set is composed of randomly generated tasks whose execution time is random in the range of 0.75–1.25 seconds. The second configuration is same as in an existing work [16] where the task set is composed of tasks from the MiBench [14].

B. Simulation results

We first compared RIF to GSR and PRS when the task set is composed of 5 randomly generated tasks (see Fig. 1). RIF-ERIS and RIF-GRIS have similar performance, and both of them achieve a lower PoF than GSR and PRS in all cases. Compared to PRS, RIF, either with ERIS or with GRIS, allocates more recoveries to tasks and the average PoF of RIF is about of 83%, 0.0009%, 0.0007%, 0.0003%, and 0.0002% of PRS for slack lengths of 1.0 s, 1.5 s, 2.0 s, 2.5 s, and 3.0 s. Compared to GSR, the average PoF of RIF is about 71%, 0.0007%, 0.0005%, 0.0008%, 48%. This simulation shows that although very short slack weakens the benefits of RIF, it achieves a lowest PoF in all cases but GSR is effective only when the slack is large.

We extended our evaluation of RIF when the workload is heavy (see Fig. 2(a)) and when a task's execution time does not follow any distributions (see Fig. 2(b)). In Fig. 2(a), the average PoF of RIF is only 75%, 0.005%, 0.004%, 0.001%, and 19% of GSR, and 91%, 0.006%, 0.005%, 0.002%, and 0.001% of PRS, respectively. In Fig. 2(b), RIF and PRS achieve similar PoFs since the execution times of some tasks are very short and both RIF and PRS can allocate recoveries. Thanks to the dynamic recovery allocation technique, RIF can

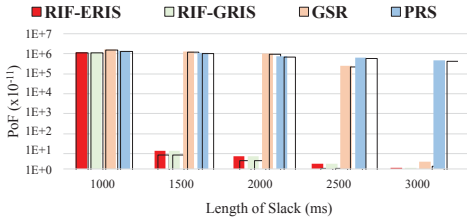
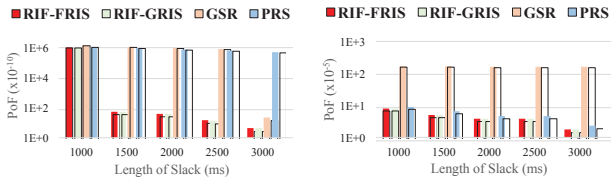


Fig. 1. PoFs when the task set has 5 randomly generated tasks.



(a) (b)

Fig. 2. PoFs when task set is composed of (a) 10 randomly generated tasks; (b) tasks from MiBench benchmark suite.

allocate recovery to tasks that have longer execution times. Both Figs. 1 and 2 show RIF is better than GSR and PRS in terms of SER improvement.

In Fig. 3, we compared RIF-ERIS and RIF-GRIS when the task set has 5 tasks and the average fault rate is large, $\lambda_L = 10^{-3}$ [16]. Based on the conditions in Theorem 1, RIF-ERIS could perform better than RIF-GRIS when the slack is small, and the PoF is reduced by about 1.15% and 0.24% when the slack is 1 s and 1.5 s, respectively. However, RIF-GRIS is better when the slack is 2 s, and 2.5 s, as RIF-GRIS reduces the PoF by about 0.98% and 2.02%, respectively. Such a difference can lead the system running without transient fault for more than 20 days. This comparison shows that although GRIS is more complicated than ERIS, it is sometime necessary and can achieve a higher SER.

VI. Conclusions

We proposed a framework to improve the system-level SER under LTR and hard real-time constraints. The SER is improved by statistically scheduling tasks and dynamically allocating recoveries. LTR is satisfied by reducing core frequencies for low priority and power-hungry tasks. Simulation results show that our approach is effective in improving SER compared to existing static recovery allocation and shared recovery allocation approaches without violating real-time and LTR constraints. As future work, we plan to extend our approach to more general task models and allowing task migration between cores.

References

[1] G. Macario, M. Torchiano, and M. Violante, "An in-vehicle infotainment software architecture based on Google Android," in *Proc. Int. Symp. Industrial Embedded Systems*, Jul. 2009, pp. 257–260.

[2] B. Zhao, H. Aydin, and D. Zhu, "Enhanced reliability-aware power management through shared recovery technology," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2009, pp. 63–70.

[3] L. Huang, F. Yuan, and Q. Xu, "Lifetime reliability-aware task allocation and scheduling on MPSoC platform," in *Proc. Design, Automation and Test in Europe*, Mar. 2009, pp. 51–56.

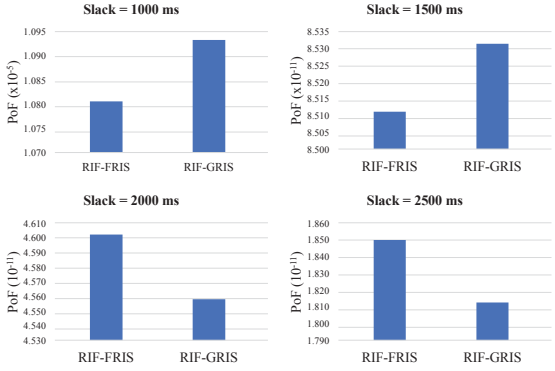


Fig. 3. Comparison of RIF-ERIS and RIF-GRIS with different lengths of slack and high fault rate.

[4] B. Zhao, H. Aydin, and D. Zhu, "Generalized reliability-oriented energy management for real-time embedded applications," in *Proc. Design, Automation Conf.*, Jun. 2011, pp. 381–386.

[5] T. Chantem, Y. Xiang, X. S. Hu, and R. P. Dick, "Enhancing multicore reliability through wear compensation in online assignment and scheduling," in *Proc. Design, Automation and Test in Europe*, Mar. 2013, pp. 1373–1378.

[6] B. Zhao, H. Aydin, and D. Zhu, "Energy management under general task-level reliability constraints," in *Proc. Int. Conf. the Real-Time and Embedded Technology and Application Symp.*, Apr. 2012, pp. 285–294.

[7] B. Nahar and B. Meyer, "Rotr: Rotational redundant task mapping for fail-operational MPSoCs," in *Proc. Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, Oct. 2015, pp. 21–28.

[8] Z. Al-bayati, B. Meyer, and H. Zeng, "Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures," in *Proc. Defect and Fault Tolerance in VLSI and Nanotechnology Systems*, Sep. 2016, pp. 57–62.

[9] A. Das, A. Kumar, B. Veeravalli, C. Bolchini, and A. Miele, "Combined DVFS and mapping exploration for lifetime and soft-error susceptibility improvement in MPSoCs," in *Proc. Design, Automation and Test in Europe*, Mar. 2014, pp. 1–6.

[10] J. Zhou, X. S. Hu, Y. Ma, and T. Wei, "Balancing lifetime and soft-error reliability to improve system availability," in *Proc. Asia and South Pacific Design Automation Conf.*, Jan. 2016, pp. 685–690.

[11] Y. Ma, T. Chantem, R. Dick, S. Wang, and S. Hu, "An on-line framework for improving reliability of real-time systems on big-little type MPSoCs," in *Proc. Design, Automation and Test in Europe*, Mar. 2017, pp. 446–451.

[12] Q. Han, M. Fan, and G. Quan, "Energy minimization for fault tolerant real-time applications on multiprocessor platforms using checkpointing," in *Proc. Int. Symp. Low Power Electronics and Design*, Aug. 2013, pp. 76–81.

[13] Nvidia, "Jetson Tegra K1." [Online]. Available: http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html

[14] Electrical Engineering and Computer Science Department, University of Michigan, "Mibench." [Online]. Available: http://vhosts.eecs.umich.edu/mibench//

[15] Y. Fu, N. Kottenstette, C. Lu, and X. D. Koutsoukos, "Feedback thermal control of real-time systems on multicore processors," in *Proc. Int. Conf. Embedded Software*, Oct. 2012, pp. 113–122.

[16] Y, Ma, et. al, "Improving system-level lifetime reliability of multicore soft real-time systems," *IEEE Trans. VLSI Systems*, vol. 25, no. 6, pp. 1895–1905, Jun. 2017.

[17] Y. Xiang, et al., "System-level reliability modeling for MPSoCs," in *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, Oct. 2010, pp. 297–306.