Compensate or Ignore? Meeting Control Robustness Requirements through Adaptive Soft-Error Handling

Kuan-Hsun Chen, Björn Bönninghoff, Jian-Jia Chen, and Peter Marwedel
Department of Informatics, Technical University of Dortmund, Germany
https://ls12-www.cs.tu-dortmund.de/
Citation: 10.1145/2907950.2907952

BIBTEX:
@inproceedings{DBLP:conf/lctrts/ChenBCM16,
  author = {Kuan{-}Hsun Chen and
            Bj{"{o}}rn B{"{o}}nninghoff and
            Jian{-}Jia Chen and
            Peter Marwedel},
  editor = {Tei{-}Wei Kuo and
            David B. Whalley},
  title = {Compensate or ignore? meeting control robustness requirements through adaptive soft-error handling},
  booktitle = {Proceedings of the 17th {ACM} {SIGPLAN/SIGBED} Conference on Languages, Compilers, Tools, and Theory for Embedded Systems, {LCTES} 2016, Santa Barbara, CA, USA, June 13 - 14, 2016},
  pages = {82--91},
  publisher = {{ACM}},
  year = {2016},
  url = {https://doi.org/10.1145/2907950.2907952},
  doi = {10.1145/2907950.2907952},
}
Compensate or Ignore? Meeting Control Robustness Requirements through Adaptive Soft-Error Handling

Kuan-Hsun Chen, Björn Böninghoff, Jian-Jia Chen, and Peter Marwedel

Abstract—To avoid catastrophic events like unrecoverable system failures on mobile and embedded systems caused by soft-errors, software-based error detection and compensation techniques have been proposed. Methods like error-correction codes or redundant execution can offer high flexibility and allow for application-specific fault-tolerance selection without the needs of special hardware supports. However, such software-based approaches may lead to system overload due to the execution time overhead. An adaptive deployment of such techniques to meet both application requirements and system constraints is desired. From our case study, we observe that a control task can tolerate limited errors with acceptable performance loss. Such tolerance can be modeled as a \((m, k)\) constraint which requires at least \(m\) correct runs out of any \(k\) consecutive runs to be correct. In this paper, we discuss how a given \((m, k)\) constraint can be satisfied by adopting patterns of task instances with individual error detection and compensation capabilities. We introduce static strategies and provide a formal feasibility analysis for validation. Furthermore, we develop an adaptive scheme that extends our initial approach with online awareness that increases efficiency while preserving analysis results. The effectiveness of our method is shown in a real-world case study as well as for synthesized task sets.

**CATEGORIES AND SUBJECT DESCRIPTORS**

C.4 [Performance of Systems]: Fault tolerance; D.4.7 [Organization and Design]: Real-time systems and embedded systems

**KEYWORDS**

Real-time and embedded systems, Fault-Tolerance, Application-aware Adaptation

I. INTRODUCTION

Due to rising integration density, low voltage operation, and environmental influences such as electromagnetic inference and radiation, mobile and embedded systems are subject to **transient faults** in the underlying hardware \([1]\), which may corrupt the correct application execution state or incur soft-errors. Depending on the types and locations, a transient fault may severely affect the execution or even ultimately prevent lead to system failures. To avoid catastrophic events like unrecoverable system failures, fault tolerant techniques can be applied at software or hardware levels exploiting redundancy to detect and eventually correct faults. The advantages of software-based approaches for error-correction codes (ECC), redundant execution, etc. \([2]\), \([3]\), \([4]\), \([5]\), \([6]\), lie in both the flexibility and application-specific assignment of techniques as well as in the non-requirement for specialized hardware. However, the additional computation incurred by such methods, e.g., redundant executions and majority-voting, can lead to 2x-3x execution time overhead in most of the cases, where the system may not be feasible due to the overloaded execution demand.

For most of the systems, the criticality of a task is typically related to the selection of the methods previously described. However, due to the (potential) inherent safety margins and noise tolerance of control tasks, a limited number of errors might be tolerable and might only downgrade control performance; however, such limited errors might not lead to an unrecoverable system state. An initial experiment demonstrates this effect for a simple LegoNXT path-tracing application. While constantly going forward, an independent decision is made for each job to “fail” and to result in the robot steering towards the outside of the track, in which light-sensors are read periodically to stay on a circular track. This error leads either to an increase of steering actions, or in the worst case to leaving the track, which is consequently marked as a failed run. From the history of the errors, we can derive the maximum number of occurring in a given window size \(k\). In Figure 1, we show the average covered distance (compared to the maximum recorded) as well as the rate at which the experiment failed in binned sets of errors per window. The window can start from any instance as a sliding-window policy. From the previous experiment, we observe that a control task can tolerate limited errors. Such errors of a control task can be modeled as a \((m, k)\) constraint which enforces a number \(m\) of correct runs out of any \(k\) consecutive instances to be correct.

In control theory literature, techniques have been proposed to aid control applications to be stable if some signal samples are delayed \([7]\), \([8]\) or dropped \([9]\), \([10]\). Using the \((m, k)\) constraint to bound the delay occurrence \([7]\), \([8]\) or even more flexible model for varying intervals \([10]\) has been studied in
the literature. Motivated by our initial experiment, we can see that the margin of tolerable errors (e.g., delayed, dropped, wrong) during task execution as the \((m, k)\) constraint allows us to exploit the availability of different protection schemes or ignore soft-errors occasionally, so that the overhead of additional handling can be greatly decreased. If faults are not crucial, adopting techniques such as interpolation, moving average, and fuzzy design can mitigate the effect of soft-errors. In case a fault results in a completely wrong result, such samples can be dropped and it is applicable to compute a new input by using the previous inputs [7], [9], [10].

In most of control systems, quality of control is the main objective. However if the system is faulty, maintaining the correctness of all executions by trivially using full Error Detection and Correction (EDAC) to each task instance can be very costly. On the other hand, the \((m, k)\) constraint to denote its robustness requirements as well as a state our problem. Then, we provide a definition of a per-task constraint may only provide a minimum acceptable control performance. Therefore, only satisfying the \((m, k)\) constraint by executing \(m\) instances with EDAC and skipping the following \(k - m\) instances is not sufficient. Our objective is to have high quality of control most of time without paying too much resource, so that the system can still be robust in the worst case. The goal of this paper is to investigate how and when to compensate, or even ignore errors, given that we can choose from different techniques and evaluate the incurred overhead. With proper run-time decisions, we can reduce the average utilization of the system, which also results in energy reduction.

One way to comply to a given \((m, k)\) constraint is to adopt static patterns that preselect the instances that are executed with EDAC to ensure reliability. For example, all the instances can be classified statically to provide guarantees on the behavior of the control loop [7]. This can be a reasonable approach for very high fault rates. However, such over-provisioning at the expense of high overheads is likely for low fault rates, as reliable execution is enforced even if the constraint would not be violated most of time. Thus, if only providing error detection while removing the overhead of the correction, we can design a run-time adaptive approach which exploits the reliable executions and is restricted to the cases where the constraint would actually be broken.

A. Contributions

In this paper, we study how to enforce the given \((m, k)\) constraints that quantify the inherent fault tolerances of periodic tasks within a control application. Different scheduling approaches are presented and analyzed. Figure. 2 illustrates an overview of our contributions, detailed as follows:

- We show that the given \((m_i, k_i)\) constraint of a task \(\tau_i\) can be achieved by preselecting the reliable instances with a static pattern. We call this Static Pattern-Based Reliable Execution (See Section IV).
- To validate the proposed approaches, we provide a sufficient schedulability test based on a multiframe task model [11] (See Section IV-B).
- We present an adaptive approach to decide the executing task versions on-the-fly by monitoring the erroneous instances with sporadic replenishment counters, such that the amount of expensive reliable instances can be greatly reduced under low soft-error rates (See Section V).
- To show the effectiveness of our approaches, we conduct extensive simulations based on synthesized task sets and a case study consisting of a practical robotic application for the resulting overhead and utilization under different strategies and error-rates (See Section VI).

II. SYSTEM MODELS

This section provides the models and notation used in this paper. First we describe the control applications, for which we state our problem. Then, we provide a definition of a per-task constraint to denote its robustness requirements as well as a generalized model of tasks with variable software-based error handling methods.

A. Control Application Model

We consider a control application has a set of control tasks \(\Gamma = \{\tau_1, \tau_2, \ldots, \tau_N\}\), in which all the tasks are independent and preemptive. For each control task, the output will be used by itself afterwards to compute the next control activity with the sampled data periodically. With the above periodic closed-loop feedback control application, we model each control task \(\tau_i\) as a periodic task which is associated to period \(T_i\), and the relative deadline of task \(\tau_i\) is characterized by \(D_i\). A control task \(\tau_i\) releases task instances (also called jobs) repeatedly by a period \(T_i\). For simplicity of presentation, we consider implicitly-deadline tasks throughout the paper, in which \(D_i\) is equal to \(T_i\) for task \(\tau_i\). To quantify the inherent tolerance of tasks to recover from previous instances that produced either none or faulty output, each task \(\tau_i\) in a control application is associated to a robustness requirement denoted by tuple \((m_i, k_i)\), where \(m_i\) and \(k_i\) are both positive integers and \(0 < m_i \leq k_i\). That is, \(m_i\) out of any \(k_i\) consecutive jobs must be correct. We assume that the robustness requirement \((m_i, k_i)\) can be given by other means analytically [8] or empirically [10].
B. Soft-Error Handling on Task Level

Tasks and resources accessed by them can be protected against soft-errors using software-based fault tolerance techniques. Depending on selected method, error detection requires introducing redundant execution, special encoding of data [12], or control-flow checking [13]. To allow for recovery, additional effort is required, e.g., increased redundancy and voters [14]. As not all errors lead to critical failures of a task, but might only deviate the output [15], selective protection can raise efficiency but reduce quality by allowing incorrect output [16]. Without restriction to a specific method, we consider tasks to be available in three versions. Applying software-based fault-tolerance, the least protected version only provides detection of errors that would affect the remaining system, but allows incorrect output values. This version is referred to as unreliable. By adding the required protection, we obtain an error-detecting version. The third version has full error detection and correction and is thus called reliable. To denote the respective versions and the resulting execution time, we use $\tau_i^u$ for the unreliable version of the task, and $c_i^u$ for its worst-case execution time (WCET). When applying an error detection technique, the task is instead denoted by $\tau_i^d$, having WCET $c_i^d$. The notation for the error-correcting version is $\tau_i^c$ with WCET $c_i^c$. Due to the rising overhead for error detection as well as for error correction, we assume that $c_i^u < c_i^d < c_i^c$ holds.

C. Schedulability and Scheduling

To schedule all the above control tasks on a uniprocessor, we assume preemptive fixed-priority scheduling, which assigns each task a unique priority level. This is widely used in the industrial practice and is also supported in most real-time operating systems. A schedule is feasible if all the tasks meet their deadline under the specified $(m_i, k_i)$ constraints. Throughout this paper, we consider the system adopts Rate-Monotonic (RM) scheduling to schedule the control tasks from the scheduling queue. All the control tasks in this paper are indexed from 1 to $N$, in which $\tau_1$ has the highest priority and $\tau_N$ has the lowest one. Hence, $T_1 \leq T_2 \leq \cdots \leq T_N$.

To test if an approach is feasible under our system model, one way is to use the utilization bound from the seminal result of Liu and Layland (L&L) [17]. In addition, the well-known time-demand analysis (TDA) developed in [18] is also applicable and tighter than L&L bound. However, both may reject many task sets which are schedulable actually if we pessimistically take the execution time of error-correcting (reliable) version to represent the worst case execution time of each task.

Although we only use RM scheduling for simplicity of presentation, the proposed approaches are not limited to RM scheduling. They can be easily extended for constrained-deadline tasks, in which $D_i \leq T_i$, and the priority assignment policy should be changed to Deadline-Monotonic.

III. PROBLEM OVERVIEW

In the following, we provide an exemplary task set to demonstrate the issues at hand. From here, we provide our problem definition to be considered in the following sections.

A. Motivational Example

Suppose that we are given two tasks $\tau_1$ and $\tau_2$ with properties as defined in Table I. To satisfy the given constraint $(m_2, k_2) = (1, 1)$, only $\tau_2^c$ is valid for execution, which requires computation time $c_2 = 5$ for each instance. Assuming transient faults occur at $t = 0$ and $t = 8$, the example in Figure 3 demonstrates execution scenarios for different compensation strategies. For simplicity of presentation, the provided diagram starts from time point $t = 0$.

If all $\tau_1$ instances are naively activated with $\tau_1^c$ to prevent any effects from soft-errors, $\tau_2$ is clearly not schedulable due to processor overload in Figure 3a. The overall system utilization is over 100% i.e., $\frac{x}{2} + \frac{5}{2} > 1$. To enforce the constraint $(m_1, k_1) = (2, 4)$, one way is to statically distribute the execution of reliable instances $\tau_1^c$ and unreliable instances $\tau_1^u$ in an alternating pattern. This static approach will be introduced as static Pattern-Based Reliable Execution in Section IV-A. As shown in Figure 3b, directly executing $\tau_1^c$ on the second and forth instances will guarantee satisfaction of the constraint $(m_1, k_1) = (2, 4)$ and avoid the processor overload even all the instances are erroneous. However, it is obvious that this approach is over-provisioning, as the fault does not occur on the second instance under this distribution of errors, in which

<table>
<thead>
<tr>
<th>Task</th>
<th>$(m_i, k_i)$</th>
<th>$c_i^u$</th>
<th>$c_i^d$</th>
<th>$c_i^c$</th>
<th>$T_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>(2, 4)</td>
<td>1</td>
<td>$1 + \epsilon$</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>$\tau_2$</td>
<td>(1, 1)</td>
<td>x</td>
<td>x</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

TABLE I: Example task set properties
the possibility of correctness is wasted. In addition, the overall utilization now is 100%, which may not be good in terms of energy-saving.

In this paper, we provide a run-time adaptive approach called Dynamic Compensation that enhances Static Pattern-Based Reliable Execution by recognizing the need to execute reliable instances dynamically instead of having a static schedule. As shown in Figure 3c, we can see that reliable execution is only activated once on the fourth instance, since satisfaction of the constraint \((m_1, k_1)\) would only be broken if an error occurs in this instance. If the fail rate of the system is low or \(k_i\) is larger than \(m_i\) greatly, the amount of expensive reliable executions can be reduced significantly in this way. However, if there is an additional fault which occurs at \(t = 4\), the above dynamic approach may be infeasible as Figure 3d.

Throughout the above example, it is not difficult to see that applying EDAC efficiently is not a trivial task. While \((m_i, k_i)\) robustness constraints need to be enforced, the schedulability of the system also needs to be considered. If EDAC can only be activated before the moment that the constraint would be broken, the resulting reduction of execution time can be utilized to save energy, which may be good to most mobile and embedded devices with the limitation of battery capacity.

### B. Problem Definition

From the above example, we state the problem addressed in this paper as follows: Suppose that we are given a set of independent and preemptive control tasks \(\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}\), where each task \(\tau_i\) is associated to an individual \((m_i, k_i)\) constraint. Each task has one unreliable version \(\tau_i^u\) without applying fault detection, one unreliable version \(\tau_i^d\) with fault detection, and one reliable version \(\tau_i^r\), where the WCETs are \(c_i^u\), \(c_i^d\), and \(c_i^r\), respectively. The objective is to efficiently utilize the processor by reducing the amount and thus the overhead of reliable instances \(\tau_i^r\) such that the system can satisfy both its hard real-time and \((m_i, k_i)\) constraints while maintaining low overall utilization without skipping any instance.

### IV. Static Pattern-Based Reliable Execution

In this section, we show how to enforce the \((m_i, k_i)\) constraints by applying \((m, k)\) static patterns to allocate the reliable executions for task \(\tau_i\). While the adopted pattern will affect the schedulability, stability, and flexibility, deciding the most suitable pattern is out-of-scope of this work. The scheduling analysis and the example are provided at the end of this section.

#### A. Static Pattern and EDAC Operation

To fully utilize the fault tolerance, it should be clear that the most efficient way is to execute the reliable version of task \(\tau_i\) only at the essential instances by which the amount of reliable jobs is equivalent to \(m_i\) for every \(k_i\) consecutive jobs for a \((m_i, k_i)\) constraint. To ease the static analysis as well as to reduce the implementation cost, we utilize the well-known concept of \((m, k)\)-patterns [19], [20] that defines a partitioning of jobs within any \(k_i\) consecutive jobs. To adopt the concept to apply to our purpose, we define the partitioning as follows:

**Definition 1:** The \((m, k)\)-pattern of task \(\tau_i\) is a binary string \(\Phi_i = \{\phi_{i,0}, \phi_{i,1}, \ldots, \phi_{i,(k_i-1)}\}\) which satisfies the following properties: \(\phi_{i,j}\) is a reliable instance if \(\phi_{i,j} = 1\) and an unreliable instance if \(\phi_{i,j} = 0\) and \(\sum_{j=0}^{k_i-1} \phi_{i,j} = m_i\). It is not difficult to see that if we can guarantee the reliable instances in \((m, k)\)-pattern are all correct, a \((m_i, k_i)\) constraint can be enforced with a static \((m, k)\)-pattern by definition. A trivial way is to directly execute the reliable version, which is called Reliable Execution (RE) for the rest of paper. However, directly applying the reliable version on each reliable instance is not the only option. Giving a try with an unreliable version before directly executing the reliable version in a same period may also be feasible to deliver the correct instances, which is called Detection and Recovery (DR). To note briefly, both static approaches for the rest of paper will be denoted as SRE and SDR, respectively.

For implementation, each control task \(\tau_i\) can use an index to point out the current instance on a \((m, k)\)-pattern \(\Phi_i\) with given \((m_i, k_i)\) constraint. When the current instance in \(\Phi_i\) is reliable, the reliable version and the unreliable version with fault detection should be executed accordingly depending upon the adopted strategy, i.e., RE or DR. In contrast (index points to an unreliable instance), the control task keep executing the unreliable version without fault detection safely. After all, \((m_i, k_i)\) constraint will be satisfied through RE or DR with a static \((m, k)\)-pattern that the number of reliable instances within window size \(k_i\) must be equal to \(m_i\).

#### B. Offline Scheduling Analysis

Due to the availability of multiple versions for each \(\tau_i\), the periodic control tasks may have different execution times depending upon the executing versions. To validate the system schedulability, we can utilize the multiframe task model proposed by Mok and Chen [11] for describing our task set. Each task can be transformed to a multiframe real-time task \(\tau_i\) with \(k_i\) frames, period \(T_i\), and an array of different execution times, i.e., \(\{c_{i,0}, c_{i,1}, \ldots, c_{i,k_i-1}\}\), in which the array of execution times for each task can be determined by given \((m_i, k_i)\)-patterns. Without loss of generality, we assume each task has at least two frames, i.e., \(k_i \geq 2\). If a task has a \((1, 1)\) constraint, we can artificially create a multiframe task with two same execution time frames.

**Definition 2:** Let \(\Psi_i(\rho)\) be the maximum of the sum of the execution times of any \(\rho\) consecutive frames of task \(\tau_i\). For brevity, we define \(\Psi_i(0) = 0\). It is also clear that \(\Psi_i(1) = \max_{j=0}^{k_i-1} c_{i,j}\) and \(\Psi_i(2) = \max_{j=0}^{k_i-1} (c_{i,j} + c_{i,((j+1) \mod k_i)})\). It is not difficult to see that \(\Psi_i(\rho)\) is equal to \(\Psi_i(\rho \mod k_i) + \left\lceil \frac{\rho}{k_i} \right\rceil \sum_{j=0}^{k_i-1} c_{i,j}\) when \(\rho > k_i\).

With the critical instant of multiframe task by Definition 5 in [11], the schedulability test of task \(\tau_q\) can be given as follows, in which there are \(q - 1\) higher-priority multiframe tasks \(\tau_1, \tau_2, \ldots, \tau_{q-1}\).
Lemma 1: Suppose that all the multiframe tasks with higher priority than \( \tau_q \) are schedulable under fixed priority scheduling on a uniprocessor, i.e., \( \tau_1, \tau_2, \ldots, \tau_{q-1} \). Multiframe task \( \tau_q \) is schedulable, if

\[
\exists t \text{ with } 0 < t \leq T_q \text{ and } \Psi_q(t) + \sum_{i=1}^{q-1} \Psi_i \left( \frac{t}{T_i} \right) \leq t. \quad (1)
\]

Proof. This directly comes from Theorem 5 and Lemma 6 by Mok and Chen in [11]. By using the definition of critical instant [11], we can ensure that the task \( \tau_q \) must be schedulable under fixed-priority assignment, if there exists a time point \( t \), where the worst case response time is less than deadline \( T_q \).

Since Lemma 1 takes all the maximum interference of higher priority jobs for task \( \tau_q \) into account, we can adopt \( \Psi_q \) to find out the maximum of the execution times among the frames of \( \tau_q \) in offline. After all, we can build a table for the first \( k_i \) entries to construct a look-up table, and derive \( \Psi_i \) in \( O(k_i^2) \) for \( \rho = 1, 2, \ldots, k_i - 1 \). To test the schedulability, all the considered frames in the test should be introduced by the worst case that all the unreliable instances are assumed to be erroneous. For those two different strategies, i.e., DR and RE, their \( \Psi_i(\rho) \) should be different, since their peak frames with the maximum execution time are different. Given pattern \( \Phi_i \), the precise rules to be defined as follows:

- **DR**: For each unreliable instance, the execution time should be calculated as \( c_i^u \) for the unreliable version without fault detection. As the worst case is re-executing \( \tau_i^u \) after \( \tau_i^d \) in the same period, each reliable instance in \( \Phi_i \) should be calculated as \( c_i^d + c_i^u \).
- **RE**: For each unreliable instance, the execution time can be set as \( c_i^u \). As the worst case is executing \( \tau_i^u \) directly, the execution time of each reliable instance in \( \Phi_i \) is \( c_i^d \).

We show the differences as defined in Table II. Assume the given pattern is E-pattern [7], task \( \tau_1 \) with \( (2, 4) \) constraint can be represented as \( \Phi_1 = \{0, 1, 0, 1\} \). For DR strategy, according to the above rule, \( \Phi_1 = \{c_1^u, c_1^d + c_1^u, c_1^d, c_1^d + c_1^u\} \). Therefore, by checking with Lemma 1, we can know that task \( \tau_2 \) is unschedulable with DR strategy. When \( t = T_2 = 10 \),

\[
\Psi_2(1) + \Psi_1 \left( \frac{10}{4} \right) > 10, \quad (2)
\]

where \( \Psi_2(1) = c_2^u \) and \( \Psi_1(3) = c_1^u + 2 \times c_1^d + 2 \times c_1^u \). For RE strategy, pattern \( \Phi_i \) can be transferred to \( \{c_i^u, c_i^d, c_i^d, c_i^d + c_i^u\} \). Again, we can test whether task \( \tau_2 \) is schedulable by Eq (3). When \( t = 8 \),

\[
\Psi_2(1) + \Psi_1 \left( \frac{8}{4} \right) \leq 8, \quad (3)
\]

where \( \Psi_2(1) = c_2^u \) and \( \Psi_1(2) = c_1^u + c_1^u \). Therefore, we know the given tasks set is schedulable with RE strategy.

V. DYNAMIC COMPENSATION

As we reveal in the motivational example, it is too pessimistic to allocate the reliable instances strictly due to the fact that soft-errors randomly happen from time to time. To mitigate the pessimism, in this section, we propose an adaptive approach, called Dynamic Compensation, to decide the executing task version on-the-fly by enhancing Static Pattern-Based Reliable Execution and monitoring the erroneous instances with sporadic replenishment counters. The idea is to execute the unreliable instances and exploit their successful executions to postpone the moment that the system will not be able to enforce \( (m_i, k_i) \) constraint, in which the resulting distribution of execution instances are still following the binary string of static patterns in the worst case. Please note that, in Dynamic Compensation we only consider version \( \tau_i^d \) for the execution of unreliable instances in order to know whether the result of unreliable version is correct or not.

A. Preprocessing

In Static Pattern-Based Reliable Execution, we only adopt the minimum amount of reliable executions to enforce \( (m_i, k_i) \) constraints without considering the positive impact of successful unreliable instances. Here we provide a proof to show that the successful executions of unreliable instances may postpone the moment to adopt the static pattern \( \Phi_i \) while enforcing the \( (m_i, k_i) \) constraint in any consecutive \( k_i \) instances.

Suppose that the static pattern \( \Phi_i \), which is a binary string, is given as the initial input. In the dynamic compensation, we can still count a successful execution of an unreliable instance as a correct run. For the simplicity of presentation, we define each successful execution of an unreliable instance as \( S \). Technically, such an \( S \) can be considered as a 1 in the binary string. But, we need to carefully handle such cases to ensure that the future instances can still satisfy the \( (m_i, k_i) \) constraint. What we propose here is to greedily postpone the adoption of the original binary string \( \Phi_i \). Therefore, this can be imagined as if some \( S \)'s are inserted into the original binary string. According to the definition, such insertions of \( S \) (or even potentially consecutive \( S \)'s) are only possible before an unsuccessful run of an unreliable instance, labeled as a 0. We follow the previous theorem to show that the above treatment can still satisfy the \( (m_i, k_i) \) constraint:

**Theorem I**: Given a control task \( \tau_i \) with a \( (m_i, k_i) \) constraint and static pattern \( \Phi_i \). If there are \( x \) successful executions of \( \tau_i^d \) as \( S \) inserting into the sequence of operations, task \( \tau_i \) can still enforce \( (m_i, k_i) \) constraint with the given pattern \( \Phi_i \) for any consecutive \( k_i \) jobs, in which \( x \geq 0 \).

Proof. We can prove this by contradiction. Suppose that the insertion of \( x \) successful executions \( S \) violate \( (m_i, k_i) \) constraint from time \( t \) to \( t + k_i - T_i \). By definition of \( (m_i, k_i) \) constraint, the total amount of successful executions and reliable jobs must be less than \( m_i \) within time interval \([t, t + k_i - T_i]\). The interval must start with an original job 0/1 or a successful execution \( S \) including \( k_i \) consecutive executions.
For $k_i$ consecutive executions, suppose there are $x$ successful executions. $x$ successful executions $S$ are inserted into the original sequence of operations, and $x$ original instances are pushed out from the time interval. For example, the original instances of $\tau_i$ can be shown as Figure 4a, in which $x$ is 2 and $(m_i,k_i) = (3,6)$. By the assumption of not satisfying $(m_i,k_i)$ constraint, the amount of reliable instances “1”’s must be less than $m_i - x$ within time interval $[t, t+k_i\cdot T_i]$. However, the successful executions $S$ can only be inserted before 0 which implies that there are only at most $x$ of “1”’s being pushed out from the time interval as shown in Figure 4(b). It means that, in time interval $[t, t+k_i\cdot T_i]$, the total amount of successful executions and reliable instances is at least $m_i$ within the time interval. Thus, we reach the contradiction. □

By Theorem 1, we know that the successful executions of unreliable instances can postpone the adoption of the static pattern $\Phi_i$ while satisfying the $(m_i,k_i)$ constraint in any consecutive $k_i$ jobs. To capture the above advantage, we adopt a set of sporadic replenishment counters to monitor the current status of fault tolerance and aid the runtime adaptation. To exploit the most amount of unreliable instances in $(m_i,k_i)$ constraint, we need to rearrange the given pattern so that the binary string starts from 0 and ends with 1, i.e., the first instance is unreliable and the last instance is reliable. After rearranging, we count the number of partitions as $p_i$, such that one partition is composed of a group of consecutive unreliable instances and a group of consecutive reliable instances. For example, given a pattern $\Phi_i = \{0,1,1,0,0,1\}$, $p_i$ is 2, since there are two partitions, i.e., $\{0,1,1\}$ and $\{0,0,1\}$. We set counter $o_{i,j} \in O_i$, and $a_{i,j} \in A_i$, where $i$ is index of tasks, $j \in \{1, \ldots , p_i\}$, and $p_i$ is the number of partitions in task $\tau_i$. Counter $o_{i,j}$ is prepared to describe the number of unreliable instances in each partition, whereas counter $a_{i,j}$ records the number of reliable instances in the static pattern $\Phi_i$. For the above pattern $\Phi_i$, the set of counters $A_i$ will be set as $\{2,1\}$, and $O_i$ will be set as $\{1,2\}$.

B. Dynamic Compensation

For each task, we prepare a mode indicator $\Pi$ to distinguish the behaviors of dynamic compensation for different status of task, i.e, $\Pi \in \{\text{tolerant, safe}\}$. If task $\tau_i$ cannot tolerate any error in the following instances, the mode indicator will be set to $\text{safe}$ and the compensation will be activated for the robustness accordingly. If it can tolerate error in the next instance, the mode indicator will be set to $\text{tolerant}$ and execute the unreliable version with fault detection. The pseudo-code is presented in Algorithm 1, and can be detailed as follows:

1. Whenever an erroneous result is observed, the current counter $o_{i,j}$ will be decreased by one unit (Lines 4-5). After $k$ instances, one unit needs to be increased back to the same counter $o_{i,j}$ (Lines 6 and 20).
2. When the current tolerance counter $o_{i,j}$ is equal to 0, now the task is required to be executed in the $\text{safe mode}$. $\ell$ is set to $o_{i,j}$ (Lines 7-9).
3. In $\text{safe mode}$, $\ell$ will be decreased iteratively. When $\ell$ is reduced to 0, the task turns back to $\text{tolerant mode}$ and update the index of partition $j$ (Lines 14-17).

Particularly, there are two different strategies (Line 13):

- **DR**: The control task will first execute unreliable version with fault detection. If there is a fault detected in the result, the system has to re-execute the instance with the reliable version immediately in the same period.
- **RE**: In $\text{safe mode}$, the control task will execute the following instances with the amount of $o_{i,j}$ of reliable versions obstinately.

Due to the flexibility of $M$ and $O$, Algorithm 1 can be adopted for any arbitrary pattern. To note briefly, both dynamic approaches for the rest of paper will be denoted as DRE and DDR, respectively. We also notice that, in the worst case, the resulting instances sequence will perform the same as Static Pattern-Based Reliable Execution as the following:

**Lemma 2**: Given static pattern $\Phi_i$, in the worst case that all the unreliable instances are erroneous, Dynamic Compensation
will follow the static pattern $\Phi_i$ to execute EDAC as Static Pattern-Based Reliable Execution accordingly.

**Proof.** This is based on the proof as Theorem 1, by taking the fact that there is no successful unreliable versions inserting to the static pattern. If there is no insertion in the binary string of static pattern $\Phi_i$, Dynamic Compensation will have the same execution sequence of instances as Static Pattern-Based Reliable Execution on allocating EDAC. □

### C. Feasibility Test

Based on Lemma 2, thus, we can directly apply the schedulability test in Section IV-B to test the feasibility for the worst case, where all unreliable instances are applying an error detection technique. For $(m, k)$ constraints, the following theorem shows that it can be satisfied by applying Algorithm 1:

**Theorem 2:** By applying Algorithm 1 with a given pattern $\Phi_i$, the control task $\tau_i$ will always enforce $(m_i, k_i)$ constraint in any consecutive $k_i$ jobs even in the worst case.

**Proof.** We can prove this property directly. Suppose that given interval of $k_i$ consecutive executions of task $\tau_i$. There must be two cases, either some of unreliable instances are correct or all the unreliable instances are never correct.

For the first case, if the output of unreliable instances are correct, by applying Algorithm 1, the system will keep execute the unreliable instance without changing the dynamic counters. By Theorem 1, we know that the correct execution of unreliable instances only postpone the adoption of static patterns $\Phi_i$, so that the amount of correct instances is at least $m_i$ and $(m_i, k_i)$ constraint is still enforced in any consecutive $k_i$ jobs instances. For the second case that all the unreliable instances are erroneous, Lemma 2 shows that Dynamic Compensation will perform as same as Static Pattern-Based Reliable Execution, which enforces $(m_i, k_i)$ constraint by given pattern $\Phi_i$. Thus, we can conclude that $(m_i, k_i)$ constraint will be satisfied by applying Algorithm 1 even in the worst case. □

### VI. Evaluation and Discussion

In this section, we use experiments to demonstrate the effectiveness of our approaches. We compare our approaches and some baseline approaches as shown in Figure 5, listed as follows:

- **Fully Robust (FR):** The system only runs the reliable versions. This is the most robust against potential errors.
- **SRE-$\Phi$:** The system directly executes a reliable version if the current instance of $\Phi$ is reliable (see Section IV).
- **SDR-$\Phi$:** The system gives a chance to execute an unreliable version with fault detection when the current instance of $\Phi$ is reliable. If any fault is detected, a reliable version is executed immediately (see Section IV).
- **DRE-$\Phi$:** By applying Algorithm 1, the system starts to execute reliable versions if the current fault tolerance counter is depleted (see Section V).
- **DDR-$\Phi$:** By applying Algorithm 1, when the tolerance counter is depleted, the system executes an unreliable version with fault detection again. If the result is not correct, a reliable version is executed immediately (see Section V).

In general, our approaches as software-based solutions can work well with the other techniques which require the bounded occurrence of delayed/dropped samples [7], [9], [10], [8]. Although the analyses of control stability with the bounded delayed/dropped samples have been studied, these existing solutions can be considered as the above baseline approaches, i.e., FR and SRE. Specifically, applying static patterns to guarantee the presence of mandatory instances in [7] can be considered as the SRE approach. Running in an open loop for each invalid sample followed by a certain number of reliable instances in [8] is also similar as SRE approach. In [9], [10], while the sample does not appear in time, the previous control value is held for the next loop, in which all the instances are fully reliable as FR strategy to prevent from soft-errors.

The evaluation is performed with two separate experiments: a case study with a practical robotic application and a simulation of synthesized task sets. For the case study, we extend a self-balancing robotic application, i.e., NXTway-gs [21], with a fault injection mechanism and apply our compensation schemes. Here all the feasible $(m, k)$ constraint are obtained by experiments in advance. We show the utilization for varying fault-rates and $(m, k)$ constraints, and determine the maximum feasible slowdown for this system. To further investigate the relationship of utilization and schedulability, we adopt Lemma 1 in the simulation experiment, to report the success ratio in terms of the schedulability for different proposed schemes and different given patterns. For the given patterns, we only apply two well-known static $(m, k)$-patterns [19], [20], which are R-pattern and E-pattern as shown in Table III.

#### A. Case Study

We consider a well-studied self-balancing application, i.e., a two wheeled mobile robot [21] on LEGO Mindstorms NXT equipped with a ARM7 microprocessor with a bootloader modified to run the nxtOSEK. There are three periodic real-time control tasks with different properties: (1) Balance Control, (2) Path Control, (3) Distance Control, which are related...
to a Gyrosopic Sensor, two Light Sensors, and an Ultrasonic Sensor respectively. The sensors sample their environment at a given rate and are connected as slaves to an I²C peripheral bus. Values are obtained by a master controller that initiates reads from the sensors.

It has been shown that this operation can be suspected to radiation-induced faults and software-based hardening is applicable [6]. While different techniques are available to harden the complete application, it lies well beyond the scope of this paper to apply and evaluate system-wide fault-tolerance that e.g. considers control-flow and memory errors. Instead, we focus on the access to the sensors, which are connected via the I²C periphery bus. While the applicability of sophisticated software fault-tolerance mechanisms has been shown for I²C implementations [6], the sensor data is crucial to the control application and thus serves our purpose.

1) Fault Injection and Task Versions: To demonstrate the system under the threat of transient faults, we use a simplified error model and define that for each independent sampling, the value may deviate from the true value with a probability \( p_{\text{fault}} \) per instance. By providing proxies to the original calls that effectively access the bus to read the sensor values, we provide an unreliable version that heuristically injects errors to the returned value. An error detecting proxy is then provided with an according overhead [6], and a reliable proxy that uses majority voting. Within these three versions of the control tasks, all calls to read the sensors are replaced with the proxies, and, for the error detection version, the comparison result is propagated to signal the success of the respective task. The execution times for each task version are profiled and shown in Table IV, along with the respective task periods in microsecond and feasible \((m_i, k_i)\) constraints, i.e., \((1, 1), (3, 10), (3, 5)\) respectively. The robustness requirements are again derived from experiments similar to Section I where the self-balancing robot needs to follow a given monitor while keeping balance, where the fault rate was kept at \( p_{\text{fault}} = 30\% \). Within the experiment, the R-Pattern is used for both dynamic and static approaches.

2) Experimental Results: In this experiment, we show the overall utilization to compare the effectiveness of our different schemes. In addition, we vary the \((m_i, k_i)\) constraint of the Path Control task to show the corresponding impact on utilization. In order to calculate the overall utilization, we monitor the number of executed instances of each task version and multiply these by the profiled execution times. In addition, we acquire a maximum utilization resulting from applying the FR scheme, which is 0.457, and serves as our baseline as it represents the overall utilization in absence of our method, and with full protection against errors. The minimum overall utilization is 0.265 and is obtained by using the unreliable version for all task instances, resulting in no protection against soft-errors.

Figure 6 presents the results for the self-balancing application described above for different \((m_i, k_i)\) constraints and varying fault rates. We observe that as the fault rate increases, the overall utilization of dynamic compensations also rises, since the requirement of reliable executions is increased within the application execution. On the other hand, we can notice that SRE-R will always be constant for a fixed \((m_i, k_i)\) constraint, as the overall utilization is deterministic by the amount of job partitions. Using SDR-R results in lower utilization, as it benefits from the dynamic reaction according to the fault distribution. When the fault rate is as low as 0.1 and the \((m_i, k_i)\) constraint equals \((3, 10)\), the probability of activating reliable executions is rare, and, hence, both dynamic compensation approaches, i.e., DRE-R and DDR-R, can closely achieve the minimum overall utilization. On the other hand, when the fault rate is as large as 0.3 and the \((m_i, k_i)\) constraint is tight, i.e., \((7, 10)\), the difference between SRE-R and both dynamic approaches is limited. We also observe that, given a tight \((m, k)\) constraint, the SDR-R approach results in lower utilization than DRE-R. While for small \(m\), SDR-R will most likely compensate for an error that

![Fig. 6: Overall Utilization after applying different approaches on Task Path, where lower is better. Two horizontal bars represent the maximum (0.457) and the minimum utilization (0.265).](image)

**Table III:** Iterations of R-patterns and E-patterns

<table>
<thead>
<tr>
<th>((m,k))</th>
<th>R-pattern</th>
<th>E-pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>(3,10)</td>
<td>0 0 0 0 0 0 1 1 1</td>
<td>0 0 0 1 1 0 1 0 1</td>
</tr>
<tr>
<td>(5,10)</td>
<td>0 0 0 0 0 1 1 1 1</td>
<td>0 1 1 0 1 0 1 0 1</td>
</tr>
<tr>
<td>(7,10)</td>
<td>0 0 1 1 1 1 1 1 1</td>
<td>0 1 1 0 1 1 0 1 1 1</td>
</tr>
</tbody>
</table>
TABLE IV: Properties of task versions in nxtOSEK-GS [21], which are associated with data sampling of Gyroscopic Sensor, Light Sensor, and Ultrasonic Sensor respectively.

<table>
<thead>
<tr>
<th>Task Name</th>
<th>m</th>
<th>k</th>
<th>Period (us)</th>
<th>Unreliable Version (us)</th>
<th>Detection Version (us)</th>
<th>Reliable Version (us)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Balance</td>
<td>1</td>
<td>1</td>
<td>4000</td>
<td>X</td>
<td>X</td>
<td>435</td>
</tr>
<tr>
<td>Path</td>
<td>3</td>
<td>10</td>
<td>1000</td>
<td>99.267</td>
<td>102.598</td>
<td>291.139</td>
</tr>
<tr>
<td>Distance</td>
<td>3</td>
<td>5</td>
<td>3000</td>
<td>99.933</td>
<td>103.93</td>
<td>173.217</td>
</tr>
</tbody>
</table>

Fig. 7: Maximum slowdown for the LegoNXT application where a worst-case schedule is still feasible.

can safely be ignored, it benefits from being able to run the unprotected task version at higher $m/k$ ratios.

3) Utilization and Feasibility: Among all the results, we can observe that DDR-R always outperforms the other approaches in reducing the overall utilization. However, recalling that DDR-R will execute a detection instance followed by a reliable instance in case of an error, the DDR-R approach will require much execution time in the worst-case, i.e., when having a sufficient amount of consecutive errors. Even though DDR-R provides more opportunities to prevent the execution of expensive reliable instances, the schedulability test for this worst-case might fail. This is especially important when considering to slow down execution by means of DVFS, e.g. to save energy, so using Equation 1, we can determine the maximum allowed slowdown where the worst-case will pass the scheduling test. The results are shown in Figure 7, where the feasible ranges for all $m > 3$ for the Path Control Task constraint collapse as the worst-cases are identical. The observation that, while showing lower utilization in the experiment, the DDR-R scheme will be harder to schedule, thus leads us to the evaluation of synthetic task sets regarding schedulability of the different approaches for varying utilization.

B. Synthesized Task Sets

We apply the UUniFast [22] method to generate a set of utilization values with the given goal. We use the approach suggested by Davis and Burns [23] to generate the task periods according to an exponential distribution. We show the result with the bounded period values from 1us to 1000us between largest and smallest periods. We define the utilization $U_i$ of multiframe task $\tau_i$ based on its peak frame. Since there are only three frame types (versions) in our studied problem, i.e., $\tau_i^u$, $\tau_i^d$, and $\tau_i^r$, we take $\tau_i^r$ as the peak frame and set its WCET as $c_i^r = c_i^r / 3$. For the other task versions, we set $c_i^u = c_i^r / 3$ and $c_i^d = c_i^r / 3$ to emulate the software-only fault detection (i.e., SWIFT+PROFiT [24]) and error recovery. The cardinality of the task sets is 10, and $k_i$ is uniformly distributed in the range $[3, 10]$. For each $k_i$, $m_i$ is set accordingly by different ratio of over all $m/k$.

Figure 8 illustrates the simulation results. It should be clear that the success ratios of the schedulability tests for the approaches (except FR) are highly dependent on the ratio $m/k$. If $m/k$ increases, the flexibility of using different protection schemes decreases. No matter which pattern the approaches use, we can observe that the maximum of the execution times $\Psi_i$ among the frames of task $\tau_i$ are really close when $m/k$ ratio is high. We can also observe that both RE approaches, i.e., SRE-R and SRE-E, perform better (with respect to the success ratio of schedulability) than the other approaches in all the simulated cases.

We can observe that the strategies using E-patterns, i.e., SRE-E and SDR-E, are always better, in terms of the success ratio, than the same strategies using R-patterns, i.e., SRE-R and SDR-R, in our simulations. The reason is from the distribution of reliable instances. As E-patterns evenly distribute the reliable instances, in general, there are less consecutive reliable instances in a strategy using E-patterns than those in the same strategy using R-patterns. Therefore, for a low priority task, the interference from the higher priority tasks under E-patterns is usually less than the case with R-patterns. When $m/k$ is high, e.g., to 0.7 or even 0.9, we can notice that both SDR schemes, i.e., SDR-R and SDR-E, are clearly inferior to the others because SDR needs to provide certain mechanisms to achieve fault detection and re-execution.

VII. CONCLUSION

While embedded systems used for control applications are liable to both hard real-time constraints and fulfillment of operational objectives, the inherent robustness of control tasks can be exploited when applying error-handling methods to deal with transient soft-errors induced by the environment. When expressing the resulting task requirement regarding correctness as a $(m, k)$ constraint, scheduling strategies based on task versions with different types of error protection become applicable. We have introduced both static- and dynamic-pattern-based approaches, each combined with two different recovery schemes. These strategies drastically reduce utilization compared to full error protection while adhering to both robustness and hard real-time constraints. To ensure the latter for arbitrary task sets, a schedulability test is provided formally. From the evaluation results, we can conclude that the average system utilization can be reduced without any significant drawbacks and be used, e.g., to save energy. This benefit can be increased with further sophistication, however, finding feasible schedules also becomes harder.
ACKNOWLEDGMENTS

This research is supported in parts by the German Research Foundation (DFG) as part of the priority program “Dependable Embedded Systems” (SPP 1500 - spp1500.itec.kit.edu). The authors thank anonymous reviewers for their suggestions on improving this paper.

REFERENCES


