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
{kuan-hsun.chen, bjoern.boenninghoff, jian-jia.chen, peter.marwedel}@tu-dortmund.de

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

1. 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. [2–6], 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.

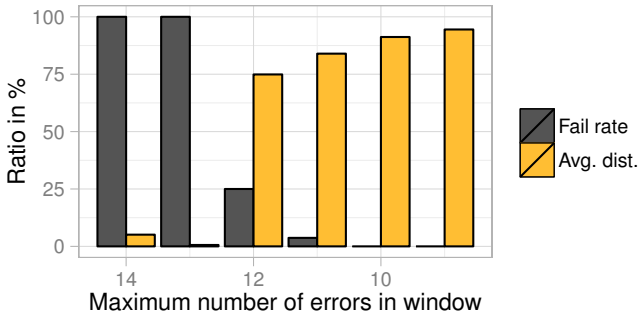


Figure 1: Avg. performance and fail rate in a LegoNXT experiment in relation to the maximum observed number of erroneous task instances for any sliding window size $k = 16$.

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

LCTES'16, June 13–14, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4316-9/16/06...$15.00
http://dx.doi.org/10.1145/2907950.2907952

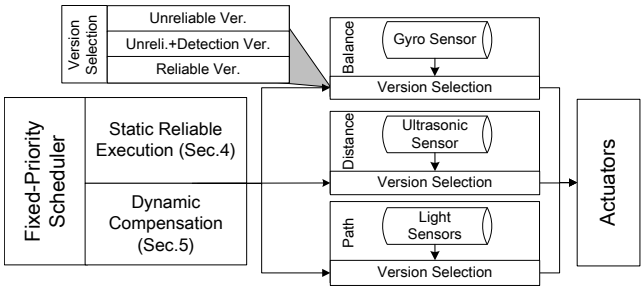


Figure 2: Overview of the considered control application

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 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.

1.1 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 τ_i can be achieved by preselecting the reliable instances with a static pattern. We call this *Static Pattern-Based Reliable Execution* (See Section 4).
- To validate the proposed approaches, we provide a sufficient schedulability test based on a multiframe task model [11] (See Section 4.2).
- 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 5).
- 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 6).

2. 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.

2.1 Control Application Model

We consider a control application has a set of control tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_N\}$ in , 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 τ_i as a periodic task which is associated to period T_i, and the relative deadline of task τ_i is characterized by D_i. A control task τ_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 τ_i. To quantify the inherent tolerance of tasks to recover from previous instances that produced either none or faulty output, each task τ_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].

2.2 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 de-

Task	(m_i, k_i)	c_i^u	c_i^d	c_i^r	T_i
τ_1	$(2,4)$	1	$1+\epsilon$	2	4
τ_2	$(1,1)$	x	x	5	8

Table 1: Example task set properties

tection 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 a 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 τ_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 τ_i^d, having WCET c_i^d. The notation for the error-correcting version is τ_i^r with WCET c_i^r. Due to the rising overhead for error detection as well as for error correction, we assume that $c_i^u < c_i^d < c_i^r$ holds.

2.3 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 τ_1 has the highest priority and τ_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.

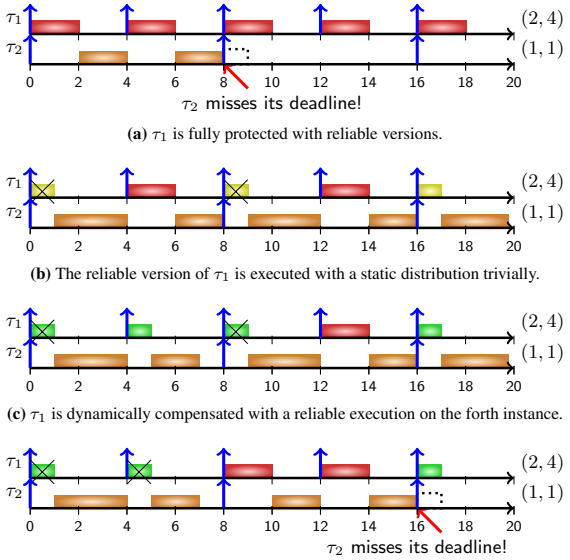
3. Problem Overview

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.

3.1 Motivational Example

Suppose that we are given two tasks τ_1 and τ_2 with properties as defined in Table 1. To satisfy the given constraint $(m_2, k_2) = (1,1)$, only τ_2^r is valid for execution, which requires computation time $c_2^r = 5$ 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 τ_1 instances are naively activated with τ_1^r to prevent any effects from soft-errors, τ_2 is clearly not schedulable due to processor overload in Figure 3a. The overall system utilization is over 100%, i.e., $\frac{2}{4} + \frac{5}{8} > 1$. To enforce the constraint (m_1, k_1) for τ_1, one way is to statically distribute the execution of reliable instances τ_1^r and unreliable instances τ_1^u in an alternating pattern. This static approach will be introduced as *Static Pattern-Based Reliable Execution* in Section 4.1. As shown in Figure 3b, directly



(a) τ_1 is fully protected with reliable versions.

(b) The reliable version of τ_1 is executed with a static distribution trivially.

(c) τ_1 is dynamically compensated with a reliable execution on the forth instance.

(d) τ_1 is not feasible if an additional fault occurring at $t = 4$.

Figure 3: Different ways to deal with soft-errors: The red block presents the reliable executions, the green block presents the executions with error detection, and the yellow block presents the unreliable version without any protection.

executing τ_1^r 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 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.

3.2 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 τ_i is associated to an individual (m_i, k_i) constraint. Each task has one unreliable version τ_i^u without applying fault detection, one unreliable version τ_i^d with fault detection, and one reliable version

τ_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 τ_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.

4. 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 τ_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.

4.1 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 τ_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 τ_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 a 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 notate briefly, both static approaches for the rest of paper will be denoted as SRE and SDR, respectively.

For implementation, each control task τ_i can use an index to point out the current instance on a (m, k)-pattern Φ_i with given (m_i, k_i) constraint. When the current instance in Φ_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.

4.2 Offline Scheduling Analysis

Due to the availability of multiple versions for each τ_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 τ_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.

Task	(m_i, k_i)	c_i^u	c_i^d	c_i^r	T_i
τ_1	$(2, 4)$	1	2	3	4
τ_2	$(1, 1)$	x	x	5	8

Table 2: Task set properties for schedulability example

DEFINITION 2. *Let $\Psi_i(\rho)$ be the maximum of the sum of the execution times of any ρ consecutive frames of task τ_i. For brevity, we define $\Psi_i(0) = 0$.*

It is also clear that $\Psi_i(1)$ is $\max_{j=0}^{k_i-1} c_{i,j}$ and $\Psi_i(2)$ is $\max_{j=0}^{k_i-1}(c_{i,j} + c_{i,((j+1) \mod k_i)})$. It is not difficult to see that $\Psi_i(\rho)$ is equal to $\Psi_i(\rho \mod k_i) + \left\lfloor \frac{\rho}{k_i} \right\rfloor \sum_{j=0}^{k_i-1} c_{i,j}$ when $\rho > k_i$.

With the critical instant of multiframe task by Definition 5 in [11], the schedulability test of task τ_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 τ_q are schedulable under fixed priority scheduling on a uniproccessor, i.e., $\tau_1, \tau_2, \ldots, \tau_{q-1}$. Multiframe task τ_q is schedulable, if*

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

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 τ_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 τ_q into account, we can adopt Ψ_q to find out the maximum of the execution times among the frames of τ_q in offline. After all, we can build a table for the first k_i entries to construct a look-up table, and derive Ψ_ρ 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 Φ_i, the precise rules to can 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 τ_i^r after τ_i^d in the same period, each reliable instance in Φ_i should be calculated as $c_i^d + c_i^r$.
- **RE:** For each unreliable instance, the execution time can be set as c_i^u. As the worst case is executing τ_i^r directly, the execution time of each reliable instance in Φ_i is c_i^r.

We show the differences as defined in Table 2. Assume the given pattern is E-pattern [7], task τ_1 with $(2, 4)$ constraint can be represented as $\Phi_1 = \{0, 1, 0, 1\}$. For DR strategy, according to the above rule, $\Phi_1 = \{c_1^u, c_1^d + c_1^r, c_1^u, c_1^d + c_1^r\}$. Therefore, by checking with Lemma 1, we can know that task τ_2 is unschedulable with DR strategy. When $t = T_2 = 10$,

$$\Psi_2(1) + \Psi_1 \left(\left\lceil \frac{10}{4} \right\rceil \right) > 10, \quad (2)$$

where $\Psi_2(1) = c_2^r$ and $\Psi_1(3) = c_1^u + 2 \times c_1^d + 2 \times c_1^r$. For RE strategy, pattern Φ_i can be transferred to $\{c_1^u, c_1^r, c_1^u, c_1^r\}$. Again,

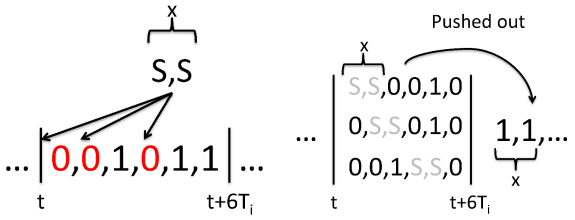


Figure 4: Example of successful executions insertion in the proof of Theorem 2, in which (a) is the original instances of τ_i and (b) is the instances of τ_i after the insertion.

we can test whether task τ_2 is schedulable by Eq (3). When $t = 8$,

$$\Psi_2(1) + \Psi_1\left(\left\lceil \frac{8}{4} \right\rceil\right) \leq 8, \tag{3}$$

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

5. 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 τ_i^d for the execution of unreliable instances in order to know whether the result of unreliable version is correct or not.

5.1 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 Φ_i while enforcing the (m_i, k_i) constraint in any consecutive k_i instances.

Suppose that the static pattern Φ_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 Φ_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 prove the following theorem to show that the above treatment can still satisfy the (m_i, k_i) constraint:

THEOREM 1. *Given a control task τ_i with a (m_i, k_i) constraint and static pattern Φ_i. If there are x successful executions of τ_i^d as S inserting into the sequence of operations, task τ_i can still enforce (m_i, k_i) constraint with the given pattern Φ_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 $t + k_i \cdot T_i$. 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 $[t, t + k_i \cdot T_i]$. 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 τ_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 Φ_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 $o_{i,j} \in \mathbb{O}_i$ and $a_{i,j} \in \mathbb{A}_i$, where i is index of tasks, $j \in \{1, \ldots, p_i\}$, and p_i is the number of partitions in task τ_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 Φ_i. For the above pattern Φ_i, the set of counters $\mathbb{A}_i$ will be set as $\{2, 1\}$, and $\mathbb{O}_i$ will be set as $\{1, 2\}$.

5.2 Dynamic Compensation

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

- 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).
- When the current tolerance counter $o_{i,j}$ is equal to 0, now the task is required to be executed in the *safe mode*. ℓ is set to $a_{i,j}$ (Lines 7-9).

Algorithm 1 Dynamic compensation of task τ_i with (m_i, k_i)

```
1:  procedure dyn_Compensation(mode Π, index j)
2:  if Π is tolerant_mode then
3:      result = execute(τ_i^d);
4:      if Fault is detected in result then
5:          o_{i,j} = o_{i,j} − 1;
6:          Enqueue_Error(o_{i,j});
7:          if o_{i,j} is equal to 0 then
8:              Set Π to safe_mode;
9:              Set ℓ to a_{i,j};
10:         end if
11:     end if
12: else
13:     either Detection_Recovery() or Reliable_Execution();
14:     ℓ = ℓ − 1;
15:     if ℓ is equal to 0 then
16:         Set Π to tolerant_mode;
17:         j = (j + 1) mod k_i;
18:     end if
19: end if
20: Update_Age(𝕆_i);
21: end procedure
```

- In *safe mode*, ℓ will be decreased iteratively. When ℓ is reduced to 0, the task turns back to *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 *safe mode*, the control task will execute the following instances with the amount of $a_{i,j}$ of reliable versions obstinately.

Due to the flexibility of $\mathbb{M}$ and $\mathbb{O}$, Algorithm 1 can be adopted for any arbitrary pattern. To notate 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 Φ_i, in the worst case that all the unreliable instances are erroneous, Dynamic Compensation will follow the static pattern Φ_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 Φ_i, Dynamic Compensation will have the same execution sequence of instances as Static Pattern-Based Reliable Execution on allocating EDAC. □

5.3 Feasibility Test

Based on Lemma 2, thus, we can directly apply the schedulability test in Section 4.2 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 Φ_i, the control task τ_i will always enforce (m_i, k_i) constraint in any consecutive k_i jobs even in the worst case.*

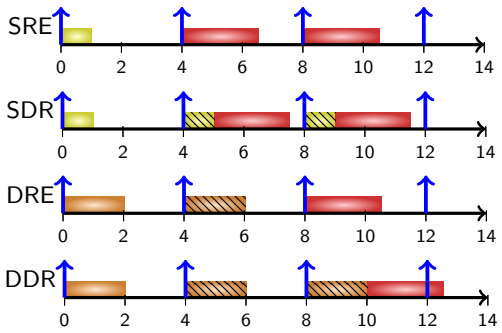


Figure 5: Example illustrates the differences. Given R-pattern (2, 3), i.e., $\Phi = (0, 1, 1)$. Suppose soft-errors happen in second and third instances. Yellow block is unreliable. Brown block is the version with detection. Red block is reliable. DDR here misses the task deadline in third instance.

Proof. We can prove this property directly. Suppose that given interval of k_i consecutive executions of task τ_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 Φ_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 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 Φ_i. Thus, we can conclude that (m_i, k_i) constraint will be satisfied by applying Algorithm 1 even in the worst case. □

6. 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-Φ: The system directly executes a reliable version if the current instance of Φ is reliable (see Section 4).
- SDR-Φ: The system gives a chance to execute an unreliable version with fault detection when the current instance of Φ is reliable. If any fault is detected, a reliable version is executed immediately (see Section 4).
- DRE-Φ: By applying Algorithm 1, the system starts to execute reliable versions if the current fault tolerance counter is depleted (see Section 5).
- DDR-Φ: 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 5).

In general, our approaches as software-based solutions can work well with the other techniques which require the bounded occurrence of delayed/dropped samples [7–10]. 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, apply-

(m,k)	R-pattern	E-pattern
(3,10)	0 0 0 0 0 0 0 1 1 1	0 0 0 1 0 0 1 0 0 1
(5,10)	0 0 0 0 0 1 1 1 1 1	0 1 0 1 0 1 0 1 0 1
(7,10)	0 0 0 1 1 1 1 1 1 1	0 1 1 0 1 1 0 1 1 1

Table 3: Iterations of R-patterns and E-patterns

ing 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 3.

6.1 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 Gyroscopic 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 I^2C 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 I^2C periphery bus. While the applicability of sophisticated software fault-tolerance mechanisms has been shown for I^2C implementations [6], the sensor data is crucial to the control application and thus serves our purpose.

6.1.1 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_{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 4, 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 1 where the self-balancing robot needs to follow a given monitor while keeping balance, where the fault rate was kept at $p_{fault} = 30\%$. Within the experiment, the R-Pattern is used for both dynamic and static approaches.

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

6.1.3 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 collate 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.

6.2 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

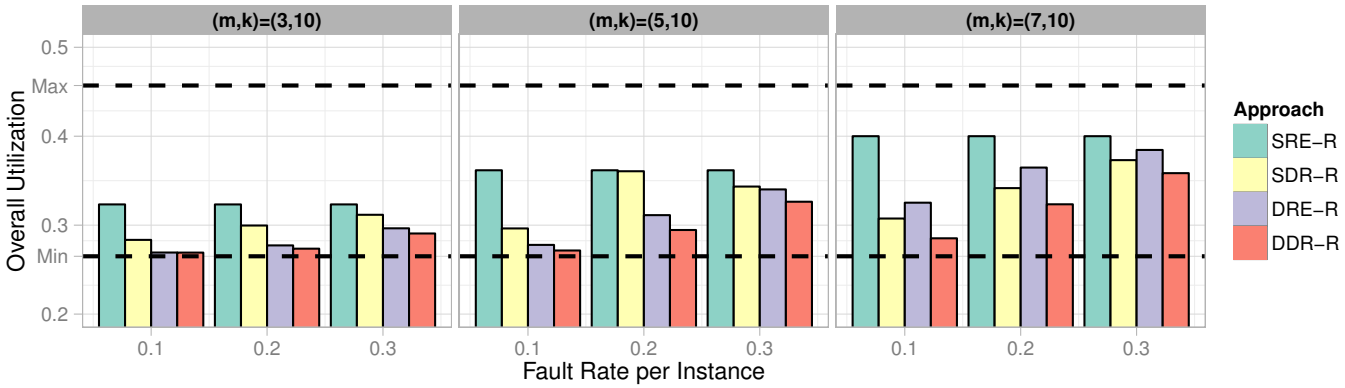


Figure 6: Overall Utilization after applying different approaches on Task Path, where lower is better. Two horizontal bars represent the maximum (0.457) and the minimum utilization (0.265).

Task Name	m	k	Period (us)	Unreliable Version (us)	Detection Version (us)	Reliable Version (us)
Balance	1	1	4000	X	X	435
Path	3	10	1000	99.267	102.598	291.139
Distance	3	5	3000	99.933	103.93	173.217

Table 4: Properties of task versions in nxtOSEK-GS [21], which are associated with data sampling of Gyroscopic Sensor, Light Sensor, and Ultrasonic Sensor respectively.

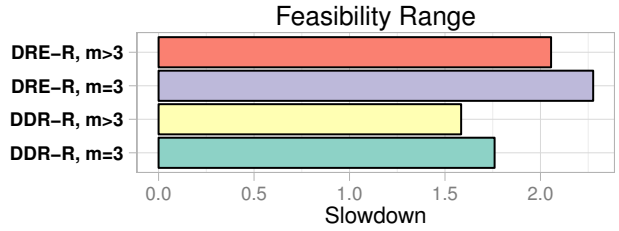


Figure 7: Maximum slowdown for the LegoNXT application where a worst-case schedule is still feasible.

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 τ_i based on its peak frame. Since there are only three frame types (versions) in our studied problem, i.e., τ_i^u, τ_i^d, and τ_i^r, we take τ_i^r as the peak frame and set its WCET as $c_i^r = T_i U_i$. For the other task versions, we set $c_i^u = c_i^r/3$ and $c_i^d = c_i^u \cdot 121\%$ 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 Ψ_i among the frames of task τ_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.

7. 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 Embed-

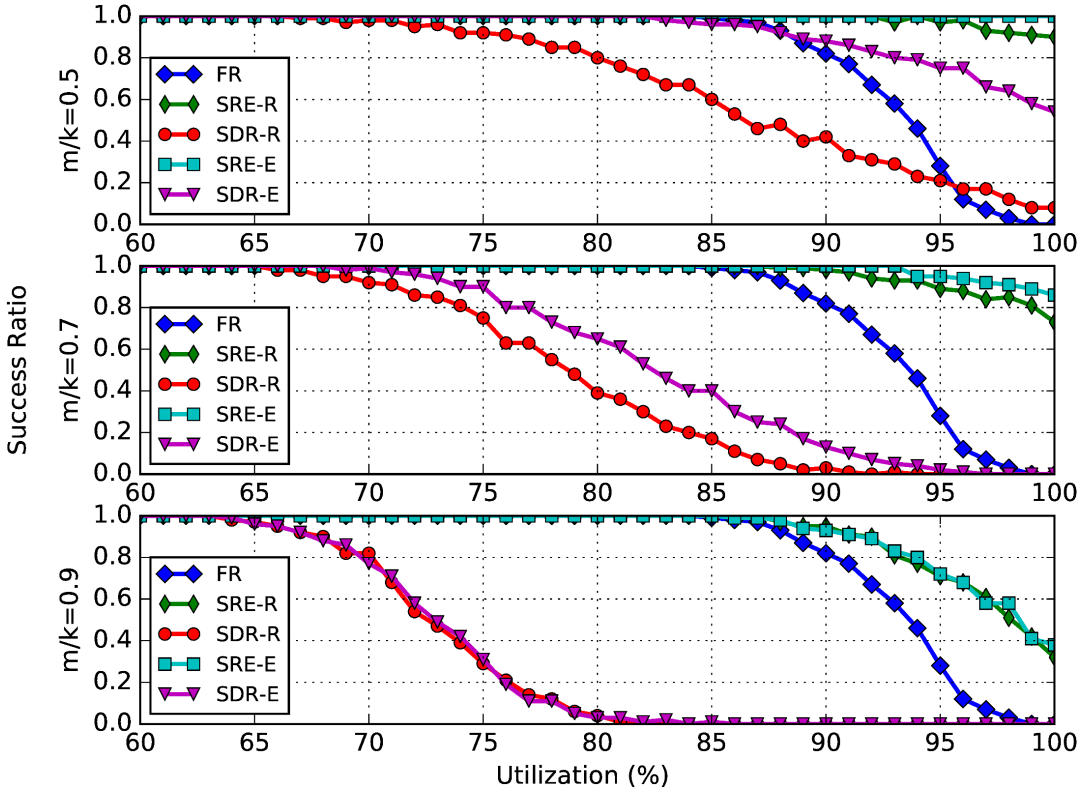


Figure 8: Success ratio comparison for different static approaches with two patterns, i.e., R-pattern and E-pattern

ded Systems" (SPP 1500 - spp1500.itec.kit.edu). The authors thank anonymous reviewers for their suggestions on improving this paper.

References

[1] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, Sept 2005.

[2] J. S. Hu, F. Li, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Compiler-directed instruction duplication for soft error detection. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 1056–1057 Vol. 2, March 2005.

[3] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, Mar 2002.

[4] S. Rehman, M. Shafique, P. V. Aceituno, F. Kriebel, J. J. Chen, and J. Henkel. Leveraging variable function resilience for selective software reliability on unreliable hardware. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1759–1764, March 2013.

[5] D. Zhu, H. Aydin, and J. J. Chen. Optimistic reliability aware energy management for real-time tasks with probabilistic execution times. In *Real-Time Systems Symposium, 2008*, pages 313–322, Nov 2008.

[6] B. Nicolescu, R. Velazco, M. Sonza-Reorda, M. Rebaudengo, and M. Violante. A software fault tolerance method for safety-critical systems: effectiveness and drawbacks. In *Integrated Circuits and Systems Design, 2002. Proceedings. 15th Symposium on*, pages 101–106, 2002.

[7] Parameswaran Ramanathan. Overload management in real-time control applications using m,k (m,k)-firm guarantee. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):549–559, June 1999.

[8] P. Kumar, D. Goswami, S. Chakraborty, A. Annaswamy, K. Lampka, and L. Thiele. A hybrid approach to cyber-physical systems verification. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 688–696, June 2012.

[9] E. Henriksson, H. Sandberg, and K. H. Johansson. Predictive compensation for communication outages in networked control systems. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, pages 2063–2068, Dec 2008.

[10] T. Bund and F. Slomka. Sensitivity analysis of dropped samples for performance-oriented controller design. In *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*, pages 244–251, April 2015.

[11] A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *Real-Time Systems Symposium, 1996., 17th IEEE*, pages 22–29, Dec 1996.

[12] Ute Schiffel, Martin Süßkraut, and Christof Fetzer. An-encoding compiler: Building safety-critical systems with commodity hardware. In *SAFECOMP '09: Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security*, pages 283–296, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 0:243–254, 2005.

[14] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 83–92, June 2006.

[15] Michael Engel, Florian Schmoll, Andreas Heinig, and Peter Marwedel. Unreliable yet useful – reliability annotations for data in cyber-physical systems. In *Proceedings of the 2011 Workshop on Software Language Engineering for Cyber-physical Systems (WS4C)*, Berlin / Germany, oct 2011.

[16] Ayswarya Sundaram, Ameen Aakel, Derek Lockhart, Darshan Thaker, and Diana Franklin. Efficient fault tolerance in multi-media applications through selective instruction replication. In *Proceedings of the*

2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies, WREFT '08, pages 339–346, New York, NY, USA, 2008. ACM.

[17] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[18] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171, Dec 1989.

[19] Gang Quan and Xiaobo Hu. Enhanced fixed-priority scheduling with (m,k)-firm guarantee. In *Proceedings of the 21st IEEE Conference on Real-time Systems Symposium*, RTSS'10, pages 79–88, Washington, DC, USA, 2000. IEEE Computer Society.

[20] Linwei Niu and Gang Quan. Energy minimization for real-time systems with (m,k)-guarantee. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(7):717–729, July 2006.

[21] Y.Yamamoto. Two wheeled self-balancing r/c robot controlled with a hitechnic gyro sensor, 2010.

[22] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2):129–154, May 2005.

[23] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers*, 57(9):1261–1276, Sept 2008.

[24] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Software-controlled fault tolerance. *ACM Trans. Archit. Code Optim.*, 2(4):366–396, December 2005.