Fault Tolerance on Control Applications: Empirical Investigations of Impacts from Incorrect Calculations

Mikail Yayla, Kuan-Hsun Chen and Jian-Jia Chen
Department of Computer Science TU Dortmund, Germany
Email: {mikail.yayla, kuan-hsun.chen, jian-jia.chen}@tu-dortmund.de

***Abstract*—Due to aggressive technology downscaling, mobile and embedded systems are susceptible to transient faults in the underlying hardware. Transient faults may incur soft-errors or even lead to system failure. A recent study has proposed to exploit the concept of the (m, k)-firm real-time task model with compensation techniques to manage redundant executions, aiming to selectively protect the control application. In this work we provide an empirical approach to find the (m, k) robustness requirements. With the delivered (m, k) robustness requirements on path tracing and balance control tasks, we conduct comprehensive case studies to evaluate the effectiveness of the compensation techniques under different fault locations and fault rates.**

I. Introduction

Mobile and embedded systems are susceptible to transient faults in the underlying hardware [1], which may occur due to rising integration density, low voltage operation, and environmental influences such as radiation and electromagnetic influences. Transient faults may alter the execution state, or incur a soft error, which is the state in which bits in memory are flipped temporarily. The consequences are difficult to predict; in the worst case they could lead to irrecoverable system failure. Recently, the Japanese satellite Hitomi crashed, because its control loop got corrupted. According to investigations in [10] a transient fault occurred in the satellite's rotation control, which led to its destruction. The financial damage was severe, a few hundred million Euros.

To protect systems from such catastrophes, one way is to apply software-based fault-tolerance techniques, such as redundant execution and error-correction code [9], [14], [12], [21]. However, trivially adopting redundant execution or error correction code may lead to significant computational overhead, e.g., when N+1 redundancy is used. Due to spatial limitation and timeliness, skillfully adopting software-based fault-tolerance approaches to prevent failure due to transient faults is not a trivial problem.

Instead of over-provisioning with extra hardware or execution time, sometimes errors can be tolerated; there may be no need to protect the whole system. Due to the potential inherent safety margins and noise interference, control applications might tolerate a limited number of errors with a downgrade of control performance without leading to an unrecoverable system state. In several studies, fault tolerance techniques have been proposed for delayed [17], [11] or dropped [8], [2] signal samples. Chen et al. [5] explored the fault tolerance of a LegoNXT path-tracing control application. They proposed to use the (m, k)-firm real-time task model to quantify the robustness of control tasks: m out of k consecutive task instances need to be correct to prevent the system from mission failure. They use this (m, k) robustness requirement to manage the expensive error correction to avoid over-provision by applying their proposed compensation techniques.

To the best of our knowledge, no detailed information to safely determine the (m, k) robustness requirement of control tasks exists in the literature. In [5], faults were only injected into sensor sampling data. Other faults, e.g., in actuators and D/A converters were not discussed. Although their results indicate that the effectiveness of the proposed compensation techniques is significant, they only considered one task at a time with fault injection instead of showing the interplay of dependent robustness requirements. To be able to apply the compensation techniques in practice, the aforementioned perspectives have to be investigated.

Our contributions:

- To investigate the impact of different fault locations on a self-balancing robot, we consider fault injection in both sensor data and motor input values in Section III.
- We present an empirical approach to find and verify (m, k) robustness requirements by analyzing the correctness of executed instances in Section IV.
- We explore impacts of concurrent fault injection with dependent (m, k) requirements in Section IV-E.
- To evaluate the overall system utilization under different fault rates and compensation techniques, we conduct comprehensive case studies in Section V.

II. System Model

This section provides the models and the notation, as well as the hard- and software used in the experiments.

A. Control Application Model

A control application has a set of periodic, preemptive control tasks, which we denote as $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task has a period T_i and a relative deadline $D_i = T_i$. The output of each instance will be used by itself again in the next instance, in a closed loop feedback control application.

A task is available in three versions with different execution times: unreliable version τ_i^u with execution time c_i^u, error detection version τ_i^d with c_i^d, and error correction version τ_i^c with c_i^c. The least protected task version is τ_i^u, which only protects from errors that lead to system crash and allows incorrect output. In the literature, several fault tolerant software techniques exist and they typically require additional

execution time for error handling, e.g., special encoding of data [20], control flow checking [18], and majority voting [3]. We assume $c_i^u < c_i^d < c_i^c$.

B. Fault Model

In this paper, we deal with potentially wrong values in the data transfer of the motor and the light sensor values caused by soft errors in the system We denote the fault rate as f throughout the paper. The probability of the occurrence of a fault is assumed, and every task instance has at most one fault under single event upset [22]. Without using error detection and correction, the system can still be executed without a system crash. However, the wrong values in the data transfer may degrade the control performance and may lead to mission failure, e.g., under soft errors the robot deviates from or leaves the track, but its operating system does not crash.

In τ_i^d, the light sensor samples twice and the two values are compared to detect an error. In τ_i^c, the sensor samples three times consecutively and returns the majority value. The frequency between consecutive samplings is high enough to produce two or three similar values with negligible difference.

To have a control of the soft errors in the control application, we assume only τ_i^u and the first execution in τ_i^d and τ_i^c versions can potentially be wrong. If the fault rate is f, then the probability that the first execution of a job is incorrect is f. The following one or two executions for detection or correction, if they exist, are assumed to be hardened perfectly; error detection and correction always perform correctly. Without such a control of errors in the experiments, the conclusions may not be drawn correctly, since the erroneous executions can be similar under different error rates.

C. (m,k) Robustness Requirement

To quantify the inherent tolerance of tasks to recover from previous instance's lack of or faulty output, we use the (m, k) robustness requirement. For each task τ_i, the robustness requirement (m_i, k_i) is given by using analytical or empirical methods; we explore the latter for each task τ_i. Any m out of k consecutive executions of a task τ_i have to be correct for the control application to run without mission failures. m_i and k_i are both positive integers and $0 < m_i \leq k_i$.

For example, consider a control task controlling specific steering actions of a vehicle. Its robustness requirement could be that it requires at least two correct in *every* three instances, denoted as $(2, 3)$. A sequence of bits contains the information whether faults were injected into instances of τ_i. When starting the system, we initialize an empty sequence of bits. If a fault was injected, we add "0" to the sequence, otherwise we add "1". This is done for all instances of τ_i. E.g., a possible sequence of bits is $\{1, 1, 0, 1, 1, 0, 1, 1\}$. In the following, we refer to this sequence of bits as bit sequence. In the experiments, the iterative way of checking is called *sliding window*, which is defined as follows: we first check the first three bits, in our example these are $\{1, 1, 0\}$, then the second three $\{1, 0, 1\}$, the third three $\{0, 1, 1\}$, etc. All such sets in the bit sequence have at least two correct task instances, we say the $(2, 3)$ requirement is fulfilled. If the $(2, 3)$ requirement is not fulfilled, e.g., $\{0, 0, 1\}$ occurs in the bit sequence, the vehicle may steer in a wrong direction and possibly cause an accident.

D. Lego NXT Robot and nxtOSEK

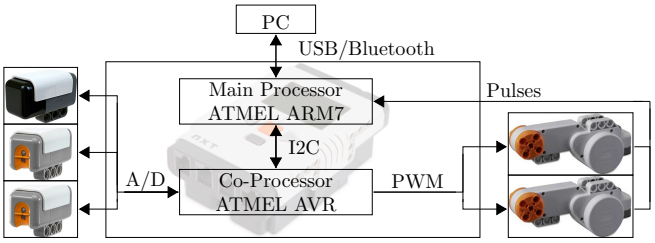

Fig. 1. The NXT system architecture. Left block: gyro sensor (top) and two light sensors (bottom two); right block: motors. Adapted from [6].

The Lego Mindstorms NXT brick is used as the computing unit for all the experiments in this paper. It runs the RTOS nxtOSEK which supports OSEK standard [15] with a C/C++ programming environment using the GCC tool chain. The C API [6] allows access to motors, sensors, and other devices, and enables the upload of C programs directly to memory. In Fig. 1, the system architecture is illustrated; more details on hardware specifications are in [19].

E. Schedulability and Scheduling

The system runs on the main CPU and adopts preemptive Rate-Monotonic Scheduling to schedule control tasks, given with fixed priorities. Task τ_1 has the highest priority and τ_n the lowest. If all tasks meet their deadlines while (m_i, k_i) holds for each τ_i, then the schedule is feasible. The schedulability test under fixed-priority scheduling is an orthogonal problem of the studied problem. For all the applied scheduling strategies, their schedulability tests can be found in [5]. For all the investigations in this work, we configure the system such that all the strategies can result in feasible schedules.

III. Studied Application with Fault Injection

This section introduces the studied control application and the fault locations. The self-balancing robot [7] uses two motors, one gyro-, and two light sensors, see Fig. 1. It has two control tasks: the path tracing task τ_{path}, and the balance task τ_{bal}. We discuss the original design's limitations in Section III-A, and explain our extensions of it in Section III-B.

A. Original Design by Chen et al. [5]

In [5], a fault injection function, defined in the nxtOSEK kernel, is based on a predefined average fault rate to simulate the occurrence of transient faults. This function generates a 32-bit unsigned random integer and compares it to a specific value which depends on the given fault rate. Only when the random integer is larger than the specific number, the correct value of the task is replaced by a faulty value.

Fig. 2.A shows the execution steps of τ_{path}. It reads the sampling data from the light sensors, and stores the data in a

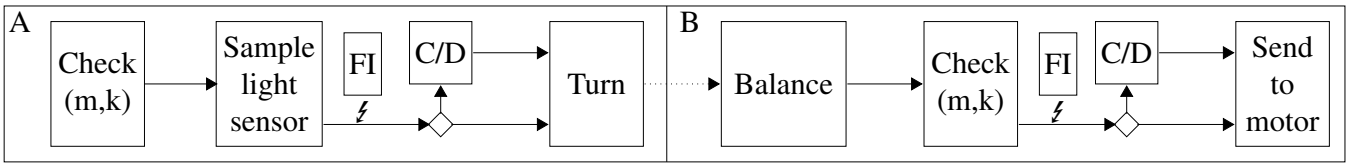

Fig. 2. The considered robotic application [7]: (A) path tracing task, (B) balance control task. Lightnings are the location of transient faults.

fuzzy table which records the most recent results of previous computations. The steering value is calculated based on the data in the table, and is referred to as "Turn" in Fig. 2. Depending on the task version, which is decided by "Check (m, k)", detection or correction (C/D) is performed. Fault injection (FI) in τ_{path} occurs after the light sensor sampling. The location of faults is illustrated with lightning symbols below FI. We consider worst-case bit flips; effects of faults are simulated as the minimum or maximum light sensor value. At the end of segment A in Fig. 2, the turn value is passed to the balance function for further calculations.

Chen et al. adopted the aforementioned mechanism to simulate the occurrence of faults and focused on how to design the compensation techniques with the given (m, k) requirement to prevent the robot from mission failures, while avoiding over-provision. After all, their approach only manipulates sensor sampling values. We extend their original fault injection to motor input values, and provide an empirical approach to find (m, k) requirements for each task.

B. Our Extended Fault Injection

In the previous study, they concluded that the balance control loop could not tolerate any single fault. However, this is not entirely true if we consider the whole task. If faults only corrupt the motor input values, and are not present inside the balance control loop, then it is possible for τ_{bal} to tolerate faults.

Thus, we inject faults into the two output values of the balance control loop, which are the motor impulses for the left and right motors; we refer to them as the motor input values. The fault injection mechanism takes the motor input value and the fault rate to determine whether to inject a fault, see Fig. 2.B. The fault value is the maximum allowed motor input value. When protection is used, the balance function is executed multiple times with the same input values as in the first execution, producing the same output values. The potentially faulty value is stored in the fuzzy table. Then the task calculates an output value on the basis of the most recent entries in the fuzzy table, delivers a final motor input value, and sends the motor input value to the corresponding motor.

IV. Empirical Approach for Obtaining (m,k) Robustness Requirements

To provide a formal empirical approach to find and verify (m, k) robustness requirements, we apply the proposed approach on τ_{path} and τ_{bal}. The goal is to verify that the robustness requirements *prevent the control application from mission failure*. The approach for finding robustness requirement candidates is proposed as follows:

The potential robustness requirement candidates (m, k) can be found by finding the minimum number i' of the correct instances, given sliding window size j, among all the possible (i, j) observed, i.e., $(m, k) = (i', k)$.

Among the candidates, the best one is decided as follows:

Out of all candidates, the best (m, k) candidate has the lowest m compared to its k, while guaranteeing the prevention of mission failure. If the ratio $\frac{m}{k}$ of (m, k) candidates is equal, the candidate with the higher difference $k - m$ is chosen.

If the ratio of k to m of two robustness requirements is equal, then (m, k) with the higher difference $k - m$ is easier to be satisfied. For example, $(1, 2)$ needs an execution pattern such as $\{0, 1, 0, 1\}$, whereas $(2, 4)$ can also allow patterns such as $\{0, 0, 1, 1\}$. We say that $(2, 4)$ has a higher flexibility than $(1, 2)$.

A. Finding and Verifying (m,k) in Practice

To clarify empirically finding and testing (m, k), we provide the instructions step-by-step. In Fig. 3, we show the steps for finding and verifying (m, k) candidates. Using these steps, our goal is to empirically prove the existence of an (m, k) requirement for a control task τ_i. In the red blocks in Fig. 3, we specify the steps for finding (m, k) candidates, and in the green blocks we specify the steps for verifying (m, k) candidates. When using numbers for the steps in this subsection, we refer to the number of the steps in Fig. 3.

In the first step, we need a fault rate for which the system always runs without mission failure, but is high enough so that any increase would cause mission failure; we refer to it as f_{max}. With f_{max}, the maximum tolerable amount of faults is injected, thus the minimum number of correct instances is executed. The knowledge about the minimum number of correct instances allows us to use protection only when necessary. Executing instances using protection costs more execution time than using no protection, so only using correction versions when necessary will minimize the cost of protection.

To start the analysis, we configure the system without fault detection and correction in the second step, so that it only executes unreliable task versions. After setting the fault rate, we run the experiment injecting faults into instances of τ_i. We record the information about whether a fault occurred in the instances of τ_i as explained in Section II-C. We do this

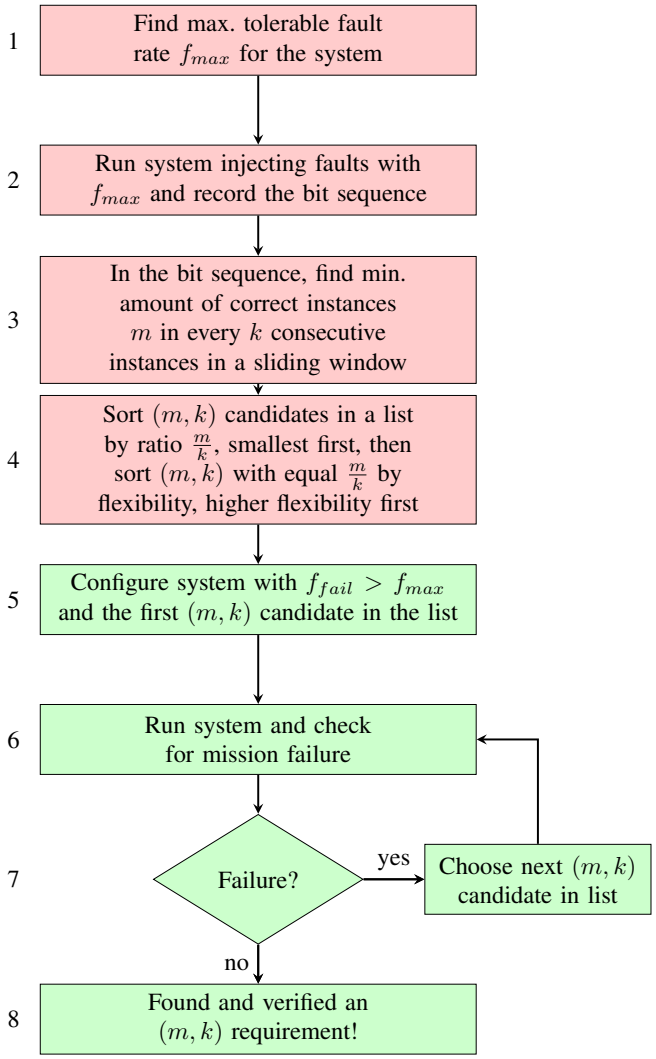

Fig. 3. The process of finding and verifying (m,k) robustness requirements. Red instructions are for finding, green ones for verifying (m,k) requirements.

for all instances of τ_i that are executed by the system in an experimental run.

In the recorded bit sequence with sufficient amount of bits, we quantify the correctness to prevent mission failure by finding the minimal amount of correct instances m in every k consecutive instances in the third step. To find (m,k) candidates, we first have to choose an interval for the sliding window size k. We use a sliding window to slide a window of size k through the bit sequence to check whether the bit sequence satisfies a certain requirement. This way, we check for the whole bit sequence whether a certain (m,k) can be observed. The goal is to find the *best* requirement.

We take the bit sequence $\{1,1,0,1,1,0,1,1\}$ as an example. We use varying sizes of sliding windows to check the recorded bits, to find potential candidates which guarantee the prevention of mission failures. By definition, $(1,1)$ is a possible candidate which always prevents mission failures, but it is pessimistic, since the correction version will always be executed. We rule out all (k,k) constraints and begin with

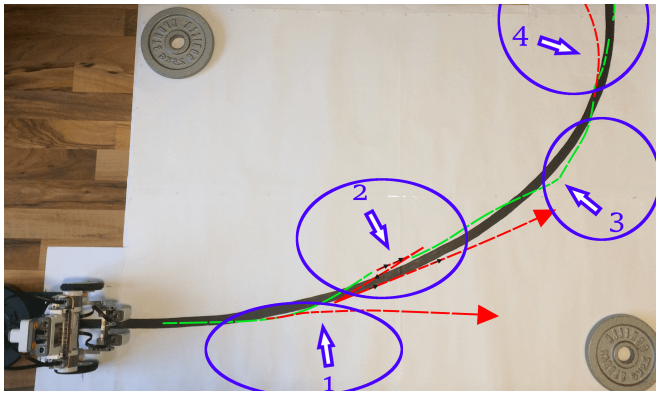

Fig. 4. Numbers on the test track correspond to the cases in Section IV-B; red lines depict failed runs, the green line represents a possible successful run.

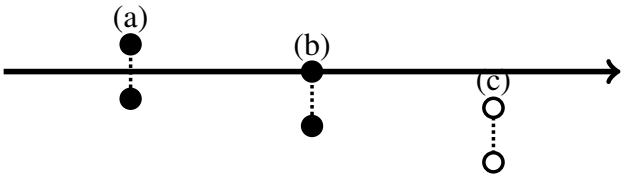

Fig. 5. Positions of the robot's light sensors on the track. In (a) and (b), the light sensors still trace the path, in (c) the sensors left the line completely.

$k=2$. For the simplicity of presentation, we stop at $k=4$.

In the bit sequence above, we observe that the minimum number of correct bits in a sliding window with $k=2$ is always one, thus we know $(1,2)$ is a potential candidate. With $k=3$, the minimum number of correct bits is two, so $(2,3)$ is also a possible candidate. For $k=4$, we gather $(2,4)$. $(3,4)$ could also be a candidate, however, it is pessimistic; we are interested in finding (m,k) with a lower $\frac{m}{k}$, so that we can execute less correction versions.

From $k=2$ to 4, we gathered $(1,2)$, $(2,3)$, and $(2,4)$ in the third step. In fourth step, we sort these candidates, beginning with the best one. We notice that the ratios of $(1,2)$ and $(2,4)$ are equal. We take $(2,4)$ first because of its higher flexibility. The list of sorted candidates is: $(2,4)$, $(1,2)$, $(2,3)$.

In the fifth step, our goal is to show that the requirements prevent mission failure, i.e., we want to verify the (m,k) requirements, the steps of which are in green in Fig. 3. We first configure the system with a fault rate $f_{fail} > f_{max}$, which causes the system to fail when no protection is used. Then, in the sixth step, we sequentially start from the first (the best) requirement in the list, which is $(2,4)$. To let the system always satisfy $(2,4)$, we use an (m,k)-pattern [16], [13], e.g., $\{0,0,1,1\}$ using R-pattern, to decide when to execute the correction version. If the system runs without mission failures when fulfilling $(2,4)$ under f_{fail}, we empirically show $(2,4)$ successfully prevents mission failure for τ_i. If there is a mission failure with $(2,4)$ in the seventh step, it may not offer enough protection, we then try the next best requirement in the list, $(1,2)$ in our case, with the pattern $\{0,1\}$, till we reach the last step.

B. Experiment Setup

The elliptical track in Fig. 4 is specified with latus rectum 105 cm, semi-latus rectum 68 cm, and path width 2 cm. We choose this quarter ellipse because it features elements of a straight and a curved line, motivated by the work in [4].

We deploy the robot on the track and choose a fault rate. The robot has two objectives: (1) balance itself, (2) trace the path. Both have to be concurrently satisfied without mission failure. If the robot falls down or leaves the track, the experimental run is counted as a failed run. If the robot leaves the track such that both light sensors are entirely out- or inside the track as illustrated in Fig. 5) (c), it is counted as a failed run. If the robot follows the line till the end, without both sensors leaving it (scenarios (a) and (b) in Fig. 5), it is counted as a successful run. If the robot leaves the track completely, but randomly returns to the track, it is counted as a failed run.

Fig. 4 shows the four possible scenarios in the experiments:

- **Case 1:** The robot drives forward and follows the first red line. Due to light sensor faults, it does not trace the path and leaves it as soon as the curvature increases.
- **Case 2:** The jagged red line is the robot's course, it occurs with faults in motor input values.The robot makes a leap forward and has to balance out that motion by driving backwards, and then it has to drive forward again to balance out the backward driving. This leads to the robot losing balance and falling down.
- **Case 3:** For the green line, sensor sampling faults lead to increased steering, in this example towards the outside of the track, but the system can recover from the errors and stay on the track. This is considered a successful run; the robot traces the line till the end.
- **Case 4:** For the last red line, the (m, k) requirement does not offer enough protection to keep the robot on the track. The robot goes off track when the curvature is the highest.

C. Bit Sequence Construction and Tracking

In the robot, the correctness of the instances is stored in one continuous bit sequence. When a fault is injected, "0" is added to the bit sequence, otherwise "1" is added. We analyze the bit sequence, find the minimal number of "1"s for a sliding window size k, and thus possible (m, k) candidates.

After the robot traces the path while recording the "1"s and "0"s constructing the bit sequence, all possible (i, j) can be found. When the robot finishes tracing, a background task starts to analyze the bit sequence and searches for minimal i for window sizes j. In our setting, the loop starts from $j = 3$. The first three bits of the bit sequence are checked, and the minimal number of "1"s is stored. Then, the following bits are analyzed with the sliding window and the minimal number of "1"s is updated. After the analysis of $(i, 3)$ is finished for the whole bit sequence, the same procedure is done with $(i, 4)$, $(i, 5)$, etc. The loop finally goes up to $j = 16$, but j can be increased manually.

Fault rate %	20	25	30	35	40
(i,j)	(5,16)	(5,16)	(5,20)	(5,20)	(5,20)

Fault rate %	10	12	14	16
(i,j)	(11,16)	(10,16)	(10,16)	(8,16)

Fig. 6. Potential (m,k) for path tracing (upper table) and balancing (lower table) in steps of 5% and 2%, respectively.

D. Finding (m, k) for Path Tracing and Balance Control

To find (m, k) requirement candidates, a version of the system without any protection is used. For τ_{path}, we tried from $f = 5\%$ to $f = 60\%$, and recorded the minimum i under given window size j in Figure 6. For the simplicity of presentation, we only present the requirement candidates under $f = 20\%$ to $f = 40\%$, the results for $f > 40\%$ do not differ from the previous ones. Under $f \geq 30\%$, $(5, 20)$ appears in all cases in steps of 1% up to $f = 52\%$, thus we consider a possible (m, k) candidate was observed. The experiment fails when $f > f_{max} = 52\%$. To the end, we choose the requirement $(5, 20)$ as our target for path tracing. For τ_{bal}, $f > f_{max} = 16\%$ causes the robot to lose balance. The potential robustness requirements are shown in Fig. 6 under different fault rates. We choose $(10, 16)$ as the candidate for τ_{bal}, as it is more stable empirically.

Interestingly, if the motor fault rate is higher than 16%, the robot makes a small leap forward and has to balance itself by driving backwards. It has to balance out that backward movement by leaping forward again, and faults may amplify that forward movement even further. This way, it increases its swaying every time, while at the same time the robot increases the amplitude of its tilt. The control loop in the balance control tries to correct the previous movement but falls down in the end. We conclude that although the injection takes place outside of the balancing application, the motor input value still has a major impact on the control loop and is able to corrupt it.

E. Verifying (m, k) for Path Tracing and Balance Control

We now select the potential requirements $(5, 20)$ for the sensors and $(10, 16)$ for the motors and test them with varying f. We assume ideal correction, thus enforcing (m, k) guarantees the correctness of m out of any k consecutive instances even under $f = 100\%$. The fault rates are always chosen such that $f > f_{max}$ of the task, so that the system has to compensate to prevent from mission failure. To enforce (m, k) requirements statically, we adopt the concept of (m, k)-patterns [16], [13]. To empirically prove that an (m, k) requirement is the minimum one, we conduct the experiments for (m, k) under different f and check if potentially better choices exist. Several runs, i.e., a minumum of 20 for one combination of (m, k) and f are conducted consecutively; if all runs are successful, it is marked as "Y", and as "N" otherwise. The observations are presented in Fig. 7 as follows:

- **Sensor task:** $(5, 20)$ always prevents mission failure under arbitrary f.

- **Motor task:** Although we found the potential requirement $(10, 16)$ previously, the robot is not stable and loses balance leading to mission failure, even under $f = 25\%$. As shown in Fig. 7, there is no one requirement that is stable for all f, except for $(16, 16)$.

(m,k)	(5,20)	(4,20)
$f = 60\%$	Y	Y
$f = 80\%$	Y	N
$f = 100\%$	Y	N

(m,k)	(16,16)	(14,16)	(11,16)	(10,16)
$f = 25\%$	Y	Y	Y	N
$f = 50\%$	Y	Y	N	N
$f = 100\%$	Y	N	N	N

Fig. 7. (m, k) tested for the path tracing task (upper table), and balance task (lower table), f is fault rate, "Y" stands for stable, "N" for mission failure.

For τ_{path}, $(4, 20)$ with $f = 60\%$ leads to a successful run, although $60\% > f_{max}$. $(4, 20)$ is very close to $(5, 20)$, which guarantees certain robustness, while 60% is also close to f_{max}, the combination of the two leads to successful runs. If the fault rate is increased, only $(5, 20)$ can guarantee successful runs. For τ_{bal}, there is no requirement satisfying all f, but $(11, 16)$ is still useful up to 25%. For sophisticated applications such as a balancer, our approach is successful up to a certain fault rate and cannot be used with arbitrarily high fault rates. Based on our empirical data, it is possible to give a guarantee that there will be no mission failure if we configure the path tracing task with $(5, 20)$. For the balancer task however, we can only give this guarantee for f up to 25%. To the end, we conclude that the robustness requirements for the τ_{path} is $(5, 20)$, and $(11, 16)$ for τ_{bal} for fault rates up to $f = 25\%$.

In reality, faults can occur in every part of the hardware. By injecting faults into motor and sensor values concurrently, we improve our model. We already verified the (m, k) requirements for τ_{path} and the τ_{bal}, therefore we again adopt the (m, k)-patterns [16], [13] with correction versions and deploy the robot with $(5, 20)$ for τ_{path} and $(11, 16)$ for τ_{bal}.

Even when enforcing $(5, 20)$ with a fault rate of 60% for the τ_{path} and $(11, 16)$ with 25% for τ_{bal}, the compensation cannot prevent the robot from mission failure, it fails in 10% of all runs by getting off track. We increased m of the two requirements minimally, and with $(8, 20)$ for τ_{path} and $(13, 16)$ for τ_{bal} the robot successfully traces the path in all runs and is stable in terms of balance. The reason comes from the dependency between τ_{path} and the τ_{bal}; both use the motors to perform their control perspectives. If we compose dependent requirements together, additional explorations are required.

V. Compensation and Overall Utilization

It is possible to derive (m, k) requirements for different tasks in the control application, and based on the given (m, k) requirements, Chen et al. [5] proposed four compensation techniques, and leverage on the concept of (m, k) pattern [16], [13], which classifies the task instances into two different types, either *reliable* marked as "1" or *unreliable* marked as "0. The compensation techniques prevent from over-provision in terms of energy consumption while satisfying the hard-real time requirement (schedulability).

Given a (m, k) pattern, we consider two significant techniques in [5] as follows:

- Static Reliable Execution (SRE): Normally the system executes τ_i^u if the current instance in the given pattern is marked as "0". By following the given pattern, the system directly executes τ_i^c when the current instance is marked as "1".
- Dynamic Detection and Recovery (DDR): The given pattern is represented as several tolerance counters, which track the number of detected faults on-the-fly. The system tolerates any faulty instances with τ_i^d but updates the counters under *tolerant mode*. When the counter is depleted, the system aggressively executes τ_i^d and compensates with τ_i^c immediately in the same period once a fault is detected under *safe mode*.

Chen et al. [5] show that DDR as an on-line adaption, dominates the other techniques in terms of overall utilization under a given configuration. We are interested whether the conclusions in [5] still hold with more configurations based on the derived (m, k) requirements in the previous section. In the following experiments, the R-Pattern [13] is used for both dynamic and static techniques, and the requirements verified in this paper are evaluated, i.e., $(5, 20)$ for τ_{path} and $(11, 16)$ for τ_{bal}.

To calculate the overall utilization, we consider the number of periods, denoted as p, and the numbers of executed unreliable, detection, and correction versions of a task, denoted as u, d, and r, respectively. The overall utilization of the compensation techniques is defined as: $\mathbb{U}_{\text{SRE}} = \frac{u \cdot c_i^u + r \cdot c_i^c}{p \cdot T_i}$ and $\mathbb{U}_{\text{DDR}} = \frac{c_i^d \cdot (d-r) + c_i^c \cdot (d+r)}{p \cdot T_i}$. The execution times are as follows: $c_i^c = 700$ μs, $c_i^d = 120$ μs, and $c_i^u = 100$ μs, which align with safety-critical systems [21], which typically use five identical instances to recover the systems from soft-errors.

The results with $(5, 20)$ requirement for τ_{path} in Fig. 8 indicate that DDR always outperforms SRE in terms of overall utilization. All experiments indicate that the margin between $\mathbb{U}_{\text{SRE}}$ and $\mathbb{U}_{\text{DDR}}$ decreases when the fault rates are higher. The most significant case is for $(5, 20)$ requirement with $f = 30\%$; the difference between $\mathbb{U}_{\text{SRE}}$ and $\mathbb{U}_{\text{DDR}}$ is extremely low with 0.04. We observe that f has more impacts on $\mathbb{U}_{\text{SRE}}$ and $\mathbb{U}_{\text{DDR}}$ than the tightness of (m, k) requirements, i.e., the difference between m and k. By setting $(11, 16)$ requirement for τ_{bal}, we observe that DDR still outperforms all the other techniques under all tests. $\mathbb{U}_{\text{DDR}}$ for $(11, 16)$ with $f = 30\%$ compared to $f = 20\%$ is only a small amount higher, since the results have similar patterns in the bit sequence.

Interestingly, if we only alter f, the difference between $\mathbb{U}_{\text{SRE}}$ and $\mathbb{U}_{\text{DDR}}$ becomes smaller, as can be seen in Fig. 8. This is because a higher f requires more compensation with τ_i^c when DDR is used. The on-line adaptions are more effective when f is lower, e.g., less than 10%, in which case we profit the most from them in the overall utilization, especially in the case

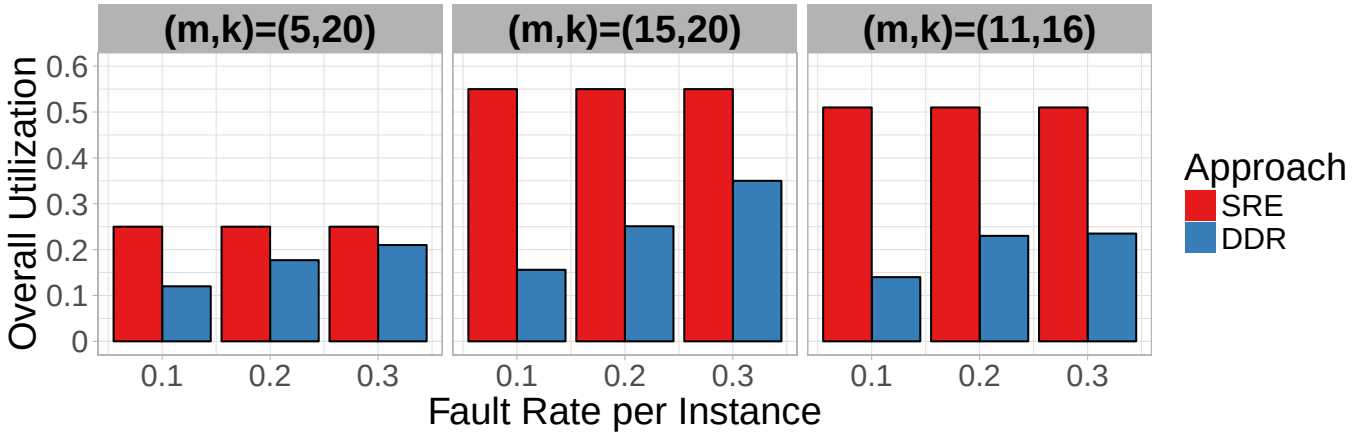

Fig. 8. Overall utilization $\mathbb{U}_{\text{SRE}}$ in red and $\mathbb{U}_{\text{DDR}}$ in blue after applying the techniques: $(5, 20)$ and $(15, 20)$ for τ_{path}, $(11, 16)$ for τ_{bal}; lower is better. The execution times are specified as $c_i^c = 700$ μs, $c_i^d = 120$ μs, and $c_i^u = 100$ μs.

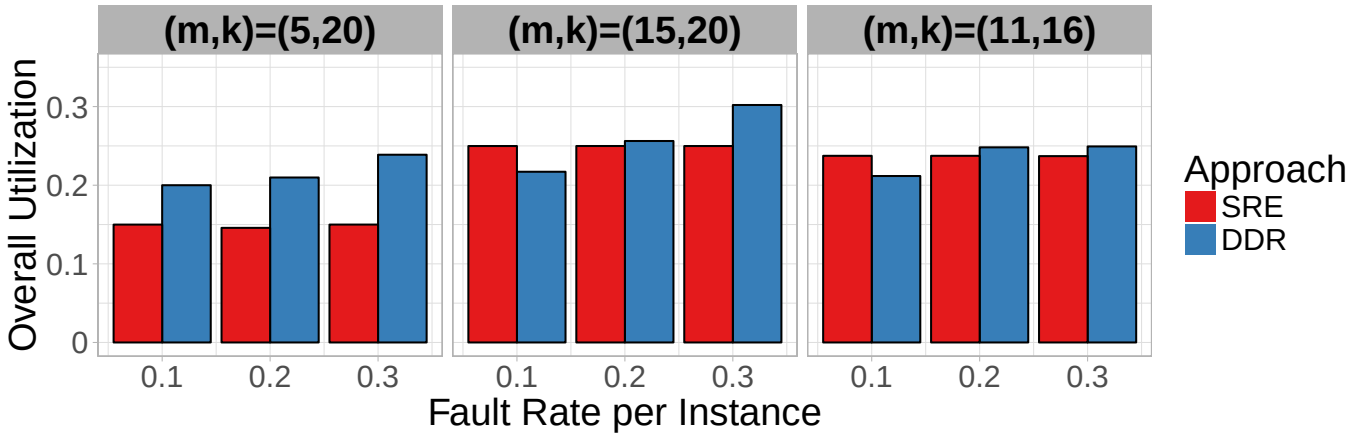

Fig. 9. Overall utilization $\mathbb{U}_{\text{SRE}}$ in red and $\mathbb{U}_{\text{DDR}}$ in blue after applying the techniques: $(5, 20)$ and $(15, 20)$ for τ_{path}, $(11, 16)$ for τ_{bal}; lower is better. The execution times are specified as $c_i^c = 300$ μs, $c_i^d = 200$ μs, and $c_i^u = 100$ μs.

of DDR. If only m is increased with a constant k, then the difference between $\mathbb{U}_{\text{SRE}}$ and $\mathbb{U}_{\text{DDR}}$ increases, because SRE executes more τ_i^c in a sliding window with higher m, while DDR can recover from faults without executing many τ_i^c.

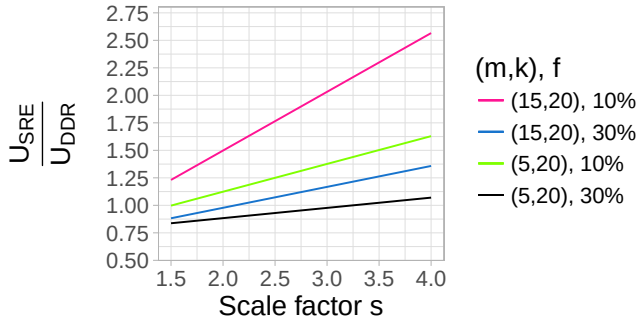

Fig. 10. The linear dependences between the scale factor s on the x axis, and the ratio $\frac{\mathbb{U}_{\text{SRE}}}{\mathbb{U}_{\text{DDR}}}$ on the y-axis describe how much DDR dominates SRE in the overall utilization.

Within the experiments, we notice that in some cases SRE as a simple off-line method can outperform the on-line adaption DDR. In Fig. 9, we specify the execution times of the system as $c_i^c = 300$ μs, $c_i^d = 200$ μs, and $c_i^u = 100$ μs, since error detection requires a minimum of two identical executions to compare the values, and error correction requires a minimum of three identical executions for majority voting. For $(5, 20)$ in Fig. 9, SRE clearly outperforms DDR, and the higher f, the more outperforming for this robustness requirement. For $(15, 20)$ and $(11, 16)$, SRE also outperforms DDR in some cases. This motivates us to investigate the impact of the difference between the correction version τ_i^c and the detection version τ_i^d by adopting a scaling factor s, i.e., $s = \frac{c_i^c}{c_i^d}$. The considered parameters here are the fault rate f, the ratio $\frac{m}{k}$ and the scaling factor s. In the results in Fig. 8, s is 5.83 and in Fig. 9, s is 1.5.

Fig. 10 presents the linear dependences between the scaling factor s and the ratio $\frac{\mathbb{U}_{\text{SRE}}}{\mathbb{U}_{\text{DDR}}}$. We present the dependences for four combinations of (m, k) and f, a tight or relaxed (m, k) requirement in tandem with lower or higher f. In all cases, we observe that the higher s, the more benefit we get from DDR. If we only increase f, the gradient decreases (see blue and black plot), since more faults force DDR to execute τ_i^d and τ_i^c within one period. If we only increase m, the gradient

increases, because when the system executes more τ_i^c, it gets more penalties with additional utilization, which increases $\mathbb{U}_{\text{SRE}}$ more than $\mathbb{U}_{\text{DDR}}$. On the lower end, we see that for $s = 1.5$, SRE only has 1.2 times overall utilization compared to DDR for the red plot, and one for the green plot. We observe a value smaller than one for the blue and black plot for $s < 2$, which means that in this region SRE outpeforms DDR, in which case $f = 30\%$.

As a consequence, we conclude that it is not always beneficial to solely adopt the on-line adaption DDR, though it seems to be the dominant technique in many cases. When f and interference noise are higher, the margin between the on-line adaption and the off-line reliable execution gets smaller; this can also be observed for more relaxed (m, k) requirements. If the system designer analyses the execution time of their execution versions τ_i^c, τ_i^d and τ_i^u and finds out that that the difference between the error correction version and the error detection version is small, e.g., $s < 2$, the off-line reliable execution could be a more beneficial choice. However, the (m, k) requirement and f need to be considered as well. Therefore, the choice between the techniques should be considered thoroughly.

VI. Conclusion

Expanding the fault injection mechanism to motor steering values, we explore the interplay among robustness requirements with fault injections. Comprehensive case studies show how the (m, k) robustness requirements can be found empirically for applying compensation techniques to avoid over-provisioning. The results indicate that the effectiveness of compensation techniques mainly depends on the execution time of correction versions and the fault rates.

To the best of our knowledge, this paper provides the first empirical approach for finding (m, k) robustness requirements on different fault locations. However, our study is still limited to control applications with least protections and perfect recovery techniques. Moreover, our soft-errors are ideally detectable and recoverable. There is also no mathematical theory yet for deriving the suitable (m, k) requirement while considering the stability for predefined controllers. We leave such a fundamental topic as future work.

Acknowledgments

This paper has been supported by DFG, as part of the Collaborative Research Center SFB876 (http://sfb876.tu-dortmund.de/), subproject B2.

References

[1] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, Sept 2005.

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

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

[4] H. Chen, Y. Song, and D. Li. Fault-tolerant tracking control of fw-steering autonomous vehicles. In *2011 Chinese Control and Decision Conference (CCDC)*, pages 92–97, May 2011.

[5] K.-H. Chen, B. Bönninghoff, J.-J. Chen, and P. Marwedel. Compensate or ignore? Meeting Control Robustness Requirements through Adaptive Soft-Error Handling. In *Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*. ACM, 2016.

[6] T. Chikamasa. nxtOSEK C API Reference. http://lejos-osek.sourceforge.net/ecrobot_c_api_frame.htm, 2013.

[7] T. Chikamasa. NXTway-GS Building Instructions. http://lejos-osek.sourceforge.net/NXTway-GS_Building_Instructions.pdf, 2013.

[8] 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.

[9] 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.

[10] J. A. E. A. (JAXA). Supplemental Handout on the Operation Plan of the X-ray Astronomy Satellite Astro-H (Hitomi). http://global.jaxa.jp/press/2016/04/files/20160428_hitomi.pdf, 2016.

[11] 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.

[12] 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 *ICSD*, 2002.

[13] L. Niu and G. Quan. Energy minimization for real-time systems with (m; k)-guarantee. *IEEE Trans. VLSI. Syst.*, 2006.

[14] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, Mar 2002.

[15] OSEK. OSEK/VDX Operating System Manual. http://www.irisa.fr/alf/downloads/puaut/TPNXT/images/os223.pdf, 2005.

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

[17] P. Ramanathan. Overload management in real-time control applications using (m, k)-firm guarantee. *IEEE Trans. Parallel Distrib. Syst.*, 1999.

[18] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: Software implemented fault tolerance. *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 0:243–254, 2005.

[19] M. Rinderknecht. Tutorial for Programming the LEGO MINDSTORMS NXT. http://www.legoengineering.com/wp-content/uploads/2013/06/download-tutorial-pdf-2.4MB.pdf, 2013.

[20] U. Schiffel, M. Süßkraut, and C. Fetzer. An-encoding compiler: Building safety-critical systems with commodity hardware. In *SAFECOMP*. Springer-Verlag, 2009.

[21] J. R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 1976.

[22] F. Wang and V. D. Agrawal. Single event upset: An embedded tutorial. In *21st International Conference on VLSI Design (VLSID 2008)*, pages 429–434, Jan 2008.