technische universität
dortmund

Bachelor Thesis

Deep Investigation for Fault Tolerance and Soft-Error Handling on Control Applications

Mikail Yayla
March 23, 2017

Advisors:
Prof. Dr. Jian-Jia Chen
M.Sc. Kuan-Hsun Chen

Faculty of Computer Science XII
Embedded Systems (LS12)
Technische Universität Dortmund
http://ls12-www.cs.tu-dortmund.de

Contents

Chapter 1

Introduction

1.1 Motivation

Mobile and embedded systems are susceptible to transient faults in the underlying hardware [1]. The reasons are rising integration density, low voltage operation and environmental influences such as radiation and electromagnetic interference. Transient faults may alter the execution state or incur soft-errors. Bitflips are a direct cause of transient faults, appear due to particle strike or interfering electromagnetic fields, and can be found in the register, processor, main memory, or other parts of a system.

The consequences of bitflips are difficult to predict, but in the worst case they may lead to catastrophic events such as unrecoverable system failure. Recently, a japanese sattelite Hitomi crashed, because its control loop got corrupted. According to investigations [2] there was a bitflip in the satellite's rotation control. This made the satellite rotate uncontrollably, leading to a crash. The financial damage was severe, since building and launching such a satellite costs a few hundred million Euros. To prevent such catastrophies, the intuitive method is to utilize software-based approaches and protect the system by using redundant execution, error-correction code, etc. [3, 4, 5, 6, 7]. Comparison or majority-voting can be used for detection and correction for example. However, redundant execution or error-correction code can lead to 2x-3x execution overhead, e.g. when N-redundancy is used, in which case N components of the system have at least one backup component. Nevertheless, due to spacial limitations and mobility of embedded systems, energy consumption and utilization should be minimized as much as possible, thus the intuitive method seems to be unsuitable.

Instead of over-provisioning with extra circuit or execution time, sometimes errors can be tolerated, so that there is no need for such a drastic measure such as redundant execution. Due to the fact that the input or output of control applications is not perfectly accurate since they are affected by the noise of the enviroment, control tasks might tolerate a limited number of errors causing only a downgrade of control performance, as opposed to leading to an unrecoverable system state. In previous studies on control applications

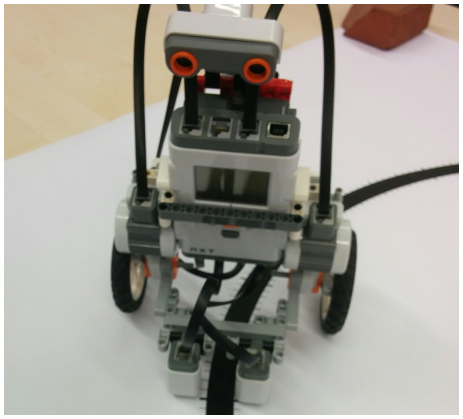

Figure 1.1: Path-tracing experiment [8]

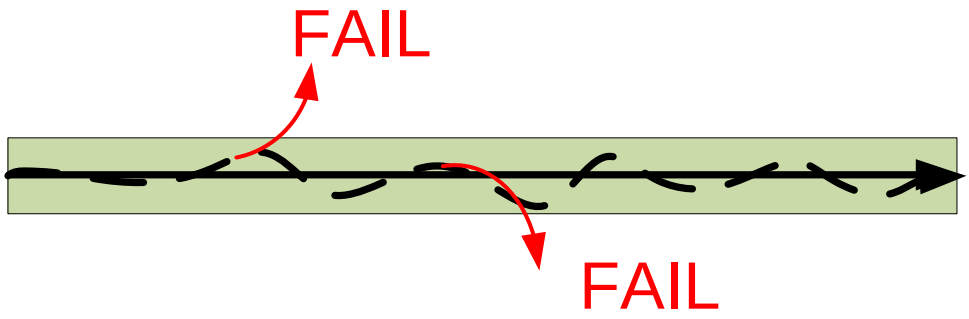

Figure 1.2: When faults are injected, there is either increased steering actions or complete fail [9]

and their tolerance against errors, techniques have been proposed for delayed [10, 11] or dropped [12, 13] signal samples. This exploration was also shown with a LegoNXT path-tracing application [8]. Its path tracing task was prepared for experiments and then executed with a fault injection mechanism, while the number of fault-free tasks was recorded to find the minimum number m of correct task instances in a sliding window size k. This robustness requirement is later referred to as (m, k), which in theory should keep the robot on the track when it is enforced on a task with error protection. While continuously going forward, the robot executed the task path-tracing. While it followed the line using light sensor information, a decision was made for each instance of the task to fail.

This error caused the robot to steer towards the outside of the track, leading to an increase in steering actions. In the worst case the robot left the track completely, without being able to find the track again. This scenario was marked as a failed run. After trying this experiment with different settings concerning the faults, the minimum number of correct instances in a given window with size k was derived.

It turns out that the path tracing task can tolerate up to ten faults in a sliding window size of $k = 16$, which is shown in Figure 1.3. Considering this phenomenon and the conclusions in previously mentioned studies about fault tolerance in control applications, we assume that a control task can tolerate a limited number of errors.

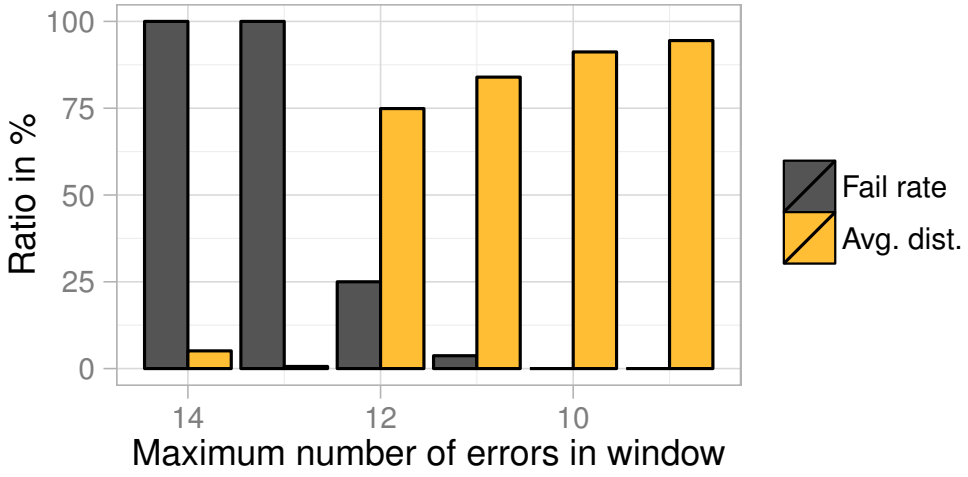

Figure 1.3: The constraint of the control application is $(6, 16)$ in a sliding window size of $k = 16$ [8]

Experiments can be run to verify if this tolerance property can be quantified with the (m, k)-firm real-time task model, which enforces m correct runs out of any k consecutive instances to be correct. The (m, k) constraint only provides a minimum acceptable control performance, thus only executing m instances correctly and skipping the next $m - k$ instances is not sufficient. The objective is to have high quality of control, e.g. a higher distance on the track, or more stable balancing behavior most of the time, without paying too much resource, additionally the system should still be robust in the worst case. The margin of tolerable errors allows for usage of different protection schemes or even ignore soft-errors occasionally in such a way, that overhead can be decreased.

It is important to examine when to compensate, or even ignore errors, while simultaniously considering overhead and quality of control. One way is to just adopt a static pattern, e.g. $\{0, 1, 1, 1\}$ and comply to $(3, 4)$ by using error protection on the tasks denoted as "1", while the other tasks indicated as "0" are executed without any protection. However, this approach would only be reasonable for very high fault rates, since m task instances would be executed with error protection using redundant executions, as a result paying execution time and overhead. Most of the time in real-life scenarios, faults happen rarely, and using the static approach would be over-provisioning. Recently, software-based run-time adaptive approaches were proposed [8] which exploit reliable executions and only follow the static pattern in cases in which the constraint would be violated. The results of [8] demonstrate that the (m, k) constraint of the path tracing task can not only be utilized to prevent the robot from leaving the track, but can also be used to decrease the overall utilization of the system by applying the newly proposed approaches.

The purpose of this thesis is to verify the results in [8] by replicating and enhancing the experiments with other experiment settings than before, and by doing so to reach more general conclusions about fault tolerance of control tasks and the proposed soft-error handling techniques.

1.2 Thesis Goals

The main goal of this thesis is to expand the research which was already done in [8] through testing the proposed techniques in different experiment setups. An answer to the following question is desired: "It worked on sensor data, but will it also work on other parts of the robot?". To confirm the results in the paper and to test if they can be used in a different spectrum, the experiments will be done with varying configurations, e.g. changing the system settings such as different forward speed, using different tracks with varying path-line widths, and magnitudes of curvature. Furthermore, fault injection will not be limited to sensor data only. In this thesis the effects of fault injection into motor steering values, the effects of concurrent fault injection into sensor and motor data, and the effectiveness of the proposed error handling techniques will be examined. Following points will be the main focus of this thesis:

- Find (m, k) constraints for different experiment setups.
- Test these (m, k) constraints and evaluate their effectiveness in terms of preventing the robot from fails.
- Examine the effectiveness of the compensation techniques with respect to overall utilization of the delivered (m, k) requirements under different fault rates.

Different Soft-Error Handling techniques, which determine the way (m, k) is enforced will be presented as well. In addition to the aforementioned points, this thesis should also provide the basic knowledge about [8], while serving as a summary of this research from a perspective of a bachelor student.

1.3 Thesis Structure

After the introduction, the background of transient faults, fault injection and control tasks are presented in Section 2. The difference between faults, errors, and system failures are also in the main focus.

Afterwards, the system model and notations, which are necessary to describe the issues [8] are introduced. Task versions, the robustness requirement (m, k), and patterns used to decide when to execute which task version are covered. Once the basics are discussed, the most important concepts in [8] are picked out as a central theme in the third section of this thesis. The soft-error handling techniques S-RE, S-DR, D-RE, and D-DR will be described in detail and their slight differences will be distinguished.

The practical part about the experiments will follow in the fourth section, the hardware and software which are used for experiments in this thesis is then presented in detail. In section five, the studied application is explored. The two control tasks and their characteristic properties are studied. The topic of the sixth section is fault injection. In that

part of the thesis, how and where faults are injected is the central point. In the following section, previous experiments concerning the error handling techniques in [8] are taken as a starting point, and taken as a guideline for further experiments. A description of the new experiment setups and the corresponding implementation follows.

The eighth section presents the collected data using figures, which will allow to display the results of the experiments. An analysis of the data will follow which will include a comparison with the experiment results in [8]. Finally, a conclusion at the end of the thesis summarizes the results of the experiments, and deduces whether the experiment results confirm the previous experiments on error handling techniques.

Chapter 2

Background

In this chapter, transient faults and their impact will be explained first. The difference between faults, errors, and system failures is particularly important. Then, an explanation of application models, task versions, and execution patterns follows as they are needed to describe soft-error handling techniques. The (m, k) constraint and its patterns play a major role in this thesis and their importance are highlighted by examples.

2.1 Causes of Transient Faults

Transient faults can occur in underlying hardware due to rising integration density, low voltage operation and environmental influences such as radiation and electromagnetic influences [1]. Through ionization, molecules and atoms which indicate the status of transistors are removed from their original place, if the radiation or electromagnetic influence is high enough [1]. This can cause a bitflip which means that a zero turns into a one or the other way around. The consequences of bitflips are very difficult to predict, in the worst case they could lead to irrecoverable system failure. This is the reason why they are so dangerous and need to be prevented, when the system executes critical tasks. By lowering the voltage operation and integration density, the chances of faults increases, because less electrons are used to distinguish the state of the hardware. Furthermore, neighboring elements in the hardware (or also neighboring systems) can influence each other as when available space on hardware gets tighter.

2.2 Transient Faults

2.2.1 Impact of Transient Faults

In some cases, transient faults have no noticeable consequences, because the flipped bit is not harmful to the system. A simple example is a scenario in which a bit was already read, the flip occurs after reading, and is ready to be overwritten. Another example is the situation in which the bitflip is not relevant for the result, such as in an OR chain for example.

In the worst case, faults cause a system failure and lead to a system crash. If a variable points to a certain location in the memory, a flipped bit in this variable may result in accessing wrong or faulty execution code, or even access to invalid memory locations, which will lead to a system crash in most cases. On the other hand, some faults may remain unnoticed. If a variable in the memory is affected and its value is manipulated, computation could proceed as planned but with faulty output values. This is called Silent Data Corruption (SDC) and will be classified in the next section.

2.2.2 Classification of Faults, Errors, and Failures

Generally, a single-bit flip in the memory hierarchy (RAM, Caches, Registers) and burst multi-bit flips count as faults, even if the faulty bit is not read by the system. If the faulty bit is read by the system, errors may occur. A detection of a fault will cause an error, specified as "true Detected Unrecoverable Error (DUE)" [14, p. 23], if the outcome of the program is affected. On the other hand, if the outcome is not affected, it will be specified as "false DUE" . However, if the outcome of the program is affected and if the fault is not detected by the system (either because protection wasn't used or it bypassed the detection mechanisms), then the scenario will count as a system failure and will be defined as Silent Data Corruption (SDC). However, not all unnoticed faults are malicious. If the outcome of the program is not affected and if there was no error protection, then there will be no error. This scenario is called "benign fault". Figure 2.1 illustrates these scenarios in a diagram.

2.2.3 Fault Detection and Correction Methods

One might argue that faults are very unlikely and one might have heard of very small numbers for the occurrence of faults, e.g. 0.044 to 0.066 FIT/Mbit for DRAM fault rate, where FIT/Mbit is the unit of measure for expected failures per 10^9 hours and 2^{20} bits [14, p. 45]. However, the "Jaguar" supercomputer at Oak Ridge Tennesee for example has approximately one failure every six hours [14, p. 45]. What's more, the decrease in size and increase in capabilities of modern hardware, their importance is expected to rise in the future. As a general rule in software based fault detection, instruction duplication is used to detect faults. A method called SWIFT proposed by Reis et al. [15] is an approach

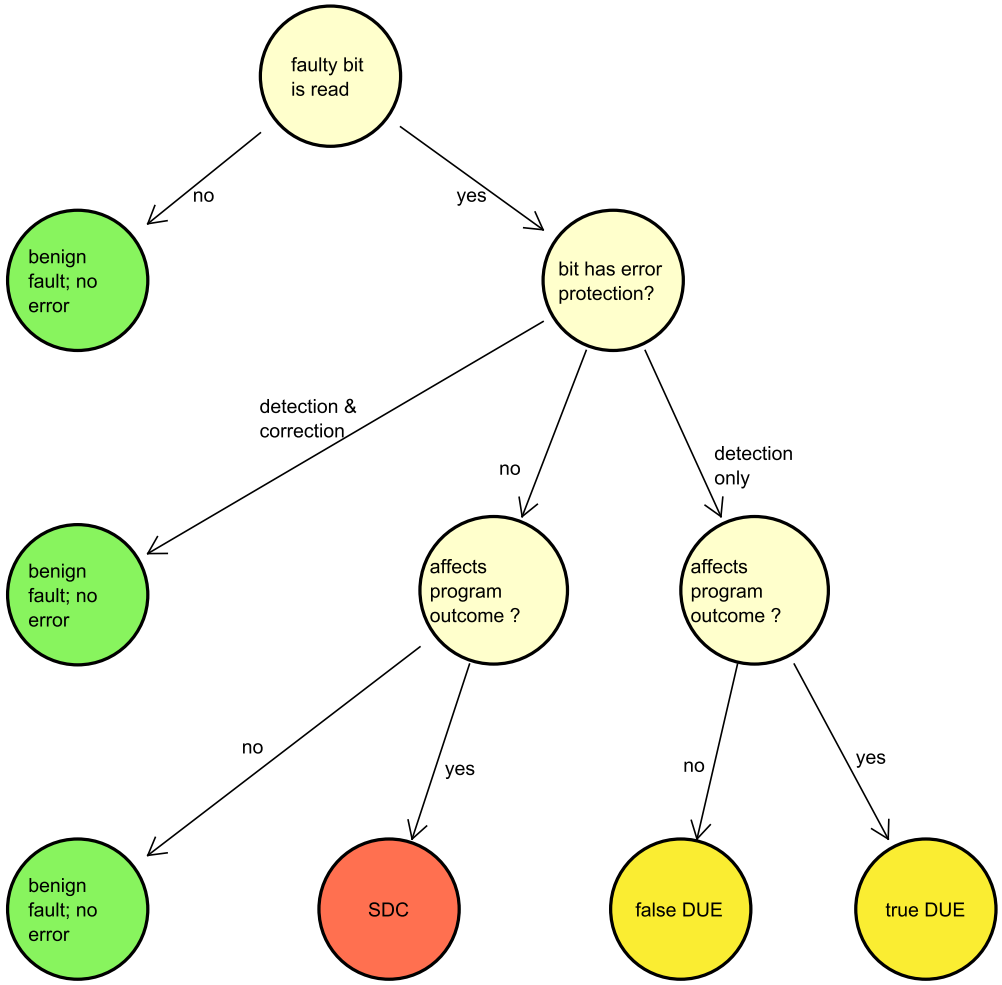

Figure 2.1: Effects of Soft-Errors, red stands for failure (SDC), yellow for DUE, and green for benign and corrected faults, adapted from [14]

which solely relies on software and can be used as a foundation for other approaches when adjustments and optimization are implemented for specific areas of application. In this thesis only software based fault detection and correction will be considered. Instruction duplication and majority voting are the two methods used in this thesis to identify faults or to correct faults respectively.

2.3 Where Faults Occur

In [8] and in this thesis, soft-error handling techniques are applied on tasks, more precisely, on instances of tasks. A task is an instance of a running program, which is being executed and managed (scheduling, memory allocation, security measures, etc.) by the operating system. Periodic tasks with a certain execution time all have the property $D_i = e_i$ and

will be used to analyze the properties of soft-error handling. An instance of a task will be called job.

In reality, faults can happen in every part of the hardware, but in this thesis and in the following experiments which model real-life scenarios, we assume that faults can only occur in sensor sampling and motor steering data, represented by bitflips in certain memory areas. Consequently these bitflips represent faults on the function body in a task. This allows us to assume that faults in our experiments only happen in some specific areas of the task instance, in this case only in those areas in which sensor and motor steering values are stored.

2.4 System Models

In this section models and notations which are used in this thesis are explained.

2.4.1 Control Application Model

A Control Application has a set of control tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ which are all independent and preemptive. The output of each task will be used by itself again in the next instance, forming a closed loop feedback control application. Each task is periodic with a period T_i and a deadline D_i, which is equal to T_i throughout the whole thesis for simplicity of presentation. The control tasks τ_i release their task instances by the period T_i. To have a measure which quantifies the inherent tolerance of tasks to recover from previous instance's lack of or faulty output, the (m_i, k_i) constraint is introduced. It measures the task's robustness against faults. m_i and k_i are both positive integers and $0 < m_i \leq k_i$. This means that m out of any k consecutive task instances have to be correct for the control application to function without errors or system failures. If the (m_i, k_i) constraint is fulfilled and a minimum of m_i out of k_i consecutive instances are correct (without errors), then there will be no noticeable effects on the system application, besides a downgrade of quality of control, such as increased steering actions in the LegoNXT experiment. However, if the constraint is not fulfilled, for example when only 9 instances are correct when the minimum allowed value m_i is equal to 10, then the system application is at danger and a successful and correct execution can not be guaranteed. In the worst case, the system may undergo system failure. It is important to note that the robustness requirement (m_i, k_i) can be given using analytical or empirical methods [8, p. 8]. Empirical methods will be used in experiments in this thesis and will be discussed in later parts of the thesis.

2.4.2 Soft-Error Handling on Task Level

Software-based fault tolerance techniques can be used to protect tasks and the resources which are accessed by them. Redundant execution, special encoding of data [16], or control flow checking [15] are required for error detection. However, if error correction or rather the

capacity to recover is desired, additional workload is required in form of increased redundancy and majority voting [17]. As indicated in the first part of the thesis, not all errors lead to system failures, but errors might merely deviate the output [18]. Thus selective protection can be applied, raising efficiency but reducing the quality of control, with the cost of having to allow deviating or in the worst case incorrect output. It is also important to note that critical system failures can be protected against through the enforcement of a robustness requirement such as (m_i, k_i). The fine-tuning of raising efficiency through selective protection and reduction of quality of control is paramount in the third chapter.

2.4.3 The Three Task Versions

Considering redundant execution and majority voting, task instances can be modeled to have three different execution times, meaning they will be available in three different versions: unreliable version τ_i^u, error detecting version τ_i^d, and error correcting or reliable version τ_i^r. The unreliable version is the least protected version, it allows incorrect output values and only protects from errors that would affect the remaining system. If error detection is used, the execution time of the task instance increases compared to the unreliable version as a matter of course, and it is called error-detecting version. To achieve error correction, the output of an instance of a task can be compared with a replica. Full error correction and detection is available if the task is executed using the third version and is thus called reliable.

As a method to correct errors, majority voting will be used. In that mechanism, the same task instance is executed three times and its results are compared. c_i is used when referring to the the worst case execution time, while τ_i is used to refer to the specific version of the task, so a task τ_i has a WCET c_i. The more protection desired, the higher the WCET of the task instance due to overhead, so we assume that $c_i^u < c_i^d < c_i^r$ holds. In Figure 2.2 below the three task versions and their relative execution times are depicted. When τ_i^d is executed, it needs two times the execution time of τ_i^u whereas τ_i^r needs three times its duration. The cases in which a replica has the same faulty result and thus errors remain unnoticed are not considered in this thesis and in the presented techniques. We assume that detection and correction always work correctly. However, we relax this assumption by allowing imperfect detection and correction, since the unreliable task versions τ_i^u are used to detect or correct errors.

2.4.4 Schedulability and Scheduling

In all cases, we assume that the system runs on an uniprocessor system and adopts Rate-Monotonic Scheduling to schedule control tasks from the scheduling queue. All control tasks have a fixed priority, in which τ_1 has the highest priority and τ_n has the lowest. Furthermore, if all the tasks meet their specific deadline while (m_i, k_i) holds for each task τ_i, then the schedule is feasible. However, it is not difficult to see that there might be

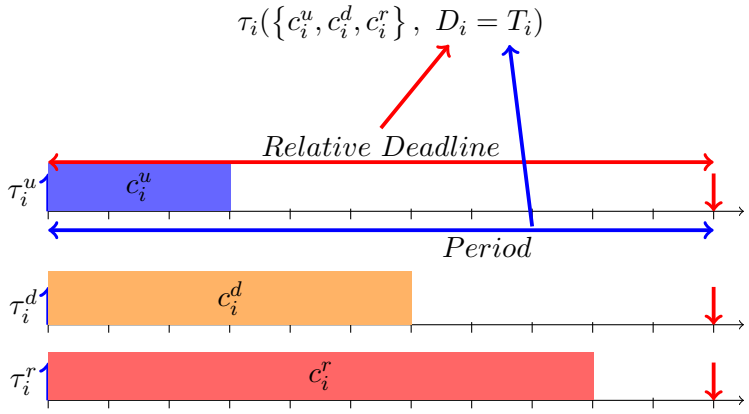

Figure 2.2: The different task versions and their execution times relative to each other [9]

problems concerning missed deadlines and other task related problems. If it is pessimistically assumed that only error correction versions of a task are used, then the schedule may not be feasible. In the following chapters methods that prevent this scenario will be presented. Nevertheless, scheduling will not be a central topic in this thesis. For further information [8, p. 4] can be referred to.

2.4.5 Selection of Task Versions

Always deciding to pick the correction version of a task instance is a possible method, but by far not a good one in terms of energy consumption and execution time. Especially if always executing the correction version is not particularly needed, because of energy consumption and the limitations on embedded systems. Control tasks can tolerate some errors at the cost of quality of control, so often times it is over-provisioning to always execute τ_i^r. All jobs of a task simply don't need to be correct, if the corresponding (m_i, k_i) constraint allows it. After ruling out the method called "Fully Robust" (run all instances with a reliable version) the question that arises is: "How do we know when to decide which version of a task to execute?". To give an answer to that question, a few examples will follow, which will motivate Pattern-Based Execution and Dynamic Compensation.

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 2.1: Example task set properties [8]

Table 2.1 shows two tasks and their characteristics. The robustness of a task is denoted by its corresponding (m_i, k_i) constraint, while the different versions of the task with their corresponding WCET can also be found in the table. τ_2 always has to execute the c_2^r version due to the $(1, 1)$ constraint, which forces correct executions and thus the activation

of error detection and correction on every task. This could be a critical task which has to be free of errors in every instance, such as the task in charge of balancing the robot.

In Figure 2.3 in the first diagram, τ_1 and τ_2 are both executed only using the τ_i^r version of the task, with full protection. This causes τ_2 to miss its deadline, since τ_1 only allows it to execute four time instances when it needs five. However, due to the $(2, 4)$ constraint of τ_1, not all jobs needs to be correct; to be specific, only two task need to be fully protected while the remaining two tasks are allowed to be erroneous. If these are erroneous, they can only cause a downgrade of quality of control. Their correctness is not checked, meaning that errors are allowed in two out of the four instances.

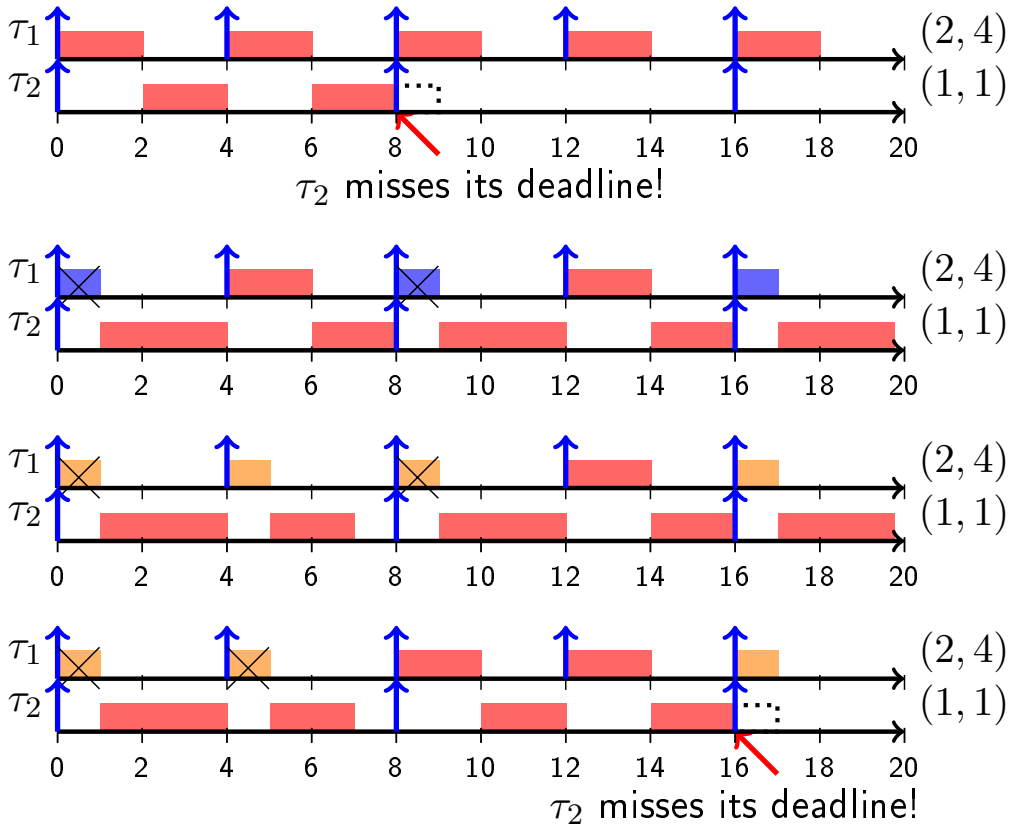

Figure 2.3: The red blocks stand for reliable executions, the orange blocks stand for executions with error detection, and the blue blocks stand for the unreliable version. The cross means that the correctness of the specific instance has no effect on (m_i, k_i). Adapted from [8].

In the second diagram in Figure 2.3 τ_1 is executed with a static distribution. In an alternating pattern τ_1 is executed without any kind of protection using τ_1^u and with full protection using τ_1^r. This allows task τ_2 to finish computing before a new period begins. A technique similar to this will be introduced as Static Pattern-Based Reliable execution. While the utilization is 100%, which is worrying in terms of energy consumption, the correctness of already executed tasks is not made use of. Making use of already executed correct task instances will be a central topic in the next chapter. The technique which

uses this approach is called Dynamic Compensation and it enhances Static Pattern-Based Reliable Execution making run-time decisions by recognizing when the reliable version of a task has to be executed to just guarantee the satisfaction of the (m_i, k_i) constraint. It is important to note that this approach always executes the detection version of a task, unless the execution of reliable versions is enforced through (m_i, k_i).

To give an introductory example, in the third diagram, the first and the third job are assumed to be erroneous. After the third instance, two instances are erroneous already and the system can't allow any more errors, thus it is forced to execute the fourth instance with full protection. Full protection is only activated on the fourth instance, since an error in it would violate the (m_i, k_i) constraint. The correctness of the second task instance and the errors on the first and third are remembered by the system in this case to make a run-time decision on the fourth instance. In the last diagram however, the first two instances have errors, which forces the system to execute the last two with full protection to satisfy (m_i, k_i), but this doesn't leave enough execution time for τ_2 and it misses its deadline on the second instance.

These examples show that applying Error Detection and Correction is not a trivial task. The (m_i, k_i) constraint has to be fulfilled while schedulability has to be considered. Fully protecting every task leads to over-provisioning and relying on static approaches potentially leads to high energy consumption while not taking advantage of already correct runs. The dynamic approach seems to do a better job at saving energy by utilizing correct runs and span (m_i, k_i) to its limit, but it may have problems with scheduling.

It is apparent that it is especially important for embedded systems to give this topic more consideration in order to prevent system failures, reduce energy consumption and utilization, while fulfilling (m_i, k_i), minimizing overhead and considering schedulability. Static Pattern-Based Reliable Execution and Dynamic Compensation will therefore be the main focus in the following parts of the thesis.

Chapter 3

Soft-Error Handling Techniques

The allocation of the reliable executions for a task τ_i to enforce (m_i, k_i) constraints using (m, k) patterns will be discussed in this section. Pattern-Based Reliable execution will be introduced first as it forms the basic concepts on which the dynamic compensation technique builds on.

3.1 Pattern Based Reliable Execution

The most efficient way to fully utilize fault tolerance is by executing the reliable version of the task only at the essential instances. In Static Pattern-Based Reliable Execution, if only m out of k instances need to be correct, then only m of those tasks need to be executed using full protection. To have a means to partition jobs and thus have an "execution plan" of when to execute which task version, the (m, k)-pattern [19, 20] is introduced.

Definition 1: The (m, k)-pattern of task τ_i is a binary string $\Phi_i = \left\{\phi_{i,0}, \phi_{i,1}, \ldots \phi_{i,(k_i-1)}\right\}$ which satisfies the following properties: $\phi_{i,j}$ is a reliable instance if $\phi_{i,j} = 1$ and a unreliable instance if $\phi_{i,j} = 0$, while $\sum_{j=0}^{k_i-1} \phi_{i,j} = m_i$ [8].

If the robustness requirement $(3, 5)$ is predefined, a corresponding (m, k) pattern would be $\Phi = \{0, 1, 0, 1, 1\}$ for example. To remember the position in the string, we assume that an index points at the corresponding value of the bitstring for implementation purposes. As for pursuing the sequence in the string, executing all instances which are denoted with a 1 with full protection and not using any protection on the instances marked with a 0 satisfies the $(3, 5)$ requirement. Directly executing the reliable versions is called Reliable Execution (RE). As the examples show in section 2.4.5, this approach is not the only option. Detection and Recovery (DR) gives a try with a fault-detecting version before executing the reliable version in the same period. In order to highlight the difference between the two variations, an example for each strategy follows. In both cases, the (m, k)-pattern $\Phi = \{0, 1, 0, 1, 1\}$ with the corresponding (m, k) constraint $(3, 5)$ will be specified for a task τ_i. Pattern-Based Reliable Execution directly executes the reliable versions whenever there

is a 1 in the bitstring and executes the unreliable version if there is a 0. $\{\tau^u, \tau^r, \tau^u, \tau^r, \tau^r\}$ would then be the schedule for this specific task. Referring to execution time, the sum of the execution times for one completion of the string is $2c^u + 3c^r$.

On the other hand, Pattern-Based Detection and Recovery first gives a try with the detection version of a task whenever the index of the pattern points to a one. If an error is detected when executing τ^d, the reliable version of the task is executed immediately after. A possible schedule may look like following if we assume that an error occurs in the third instance: $\{\tau^u, \tau^d + \tau^r, \tau^u, \tau^d, \tau^d\}$. The sum of the execution times in the worst case is $2c^u + 3(c^d + c^r)$ because errors occur every time the system gives a try with the detection version; it has to follow it up with the reliable version immediately. Figure 3.1 shows the reliable execution in the first diagram. The system just follows the pattern, and if an error occurs in one instance, then it will either be corrected by the reliable versions, or the error will be tolerated, because the (m, k) constraint is always fulfilled. In the second diagram, the system follows the pattern as well, but always gives a try with the detection version first if the bit in the pattern is a 1. There is an error in the second instance, so the system has to execute the reliable version afterwards immediately.

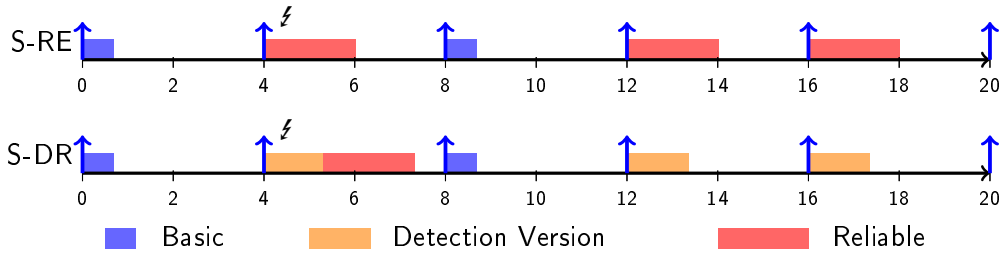

Figure 3.1: Examples for the static approaches for $(3, 5)$ with the pattern $\{0, 1, 0, 1, 1\}$, adapted from [9]

Especially when executing DR, there could be concerns about the execution time and the schedulability of tasks, because executing $\tau^d + \tau^r$ seems a lot. For one thing, there are tools to validate the system schedulability. Using the multiframe task model which is proposed by Mok and Chen [21] allows for transformation of each task to a multiframe real-task τ_i with k_i frames, period T_i and an array of different execution times, through which a task "slides". Assuming execution times $c_i^0 = 1$, $c_i^1 = 4$, $c_i^2 = 4$ as the frames, the array for task τ_i would be a set, such as $\{1, 4, 4\}$ and the task would have an execution time of one on the first and four on the second and third instance. In this case k_i is equal to three, so the (m_i, k_i) constraint determines the pattern. This model allows to set up a formula $Y_i(p)$ which calculates the maximum of the sum of the execution times of any p consecutive frames of a task τ_i through which an schedulability test can be carried out. For more information on schedulability [8, p. 4] can be referred to.

3.2 Dynamic Compensation

Soft-errors happen randomly from time to time, and the likelihood of their occurrence is generally not very high. As the previous examples show, the pattern based static techniques are too pessimistic as they allocate and execute the task versions as if soft-errors were more likely as they are in reality. Minimizing the executions of τ^r tasks is one of the main goals in this approach, as τ^r is very costly considering execution time and energy consumption. We only want to execute τ^r if it is absolutely necessary.

Dynamic Compensation enhances Static Pattern-Based Reliable Execution by making decisions on-the-fly. The main idea is to postpone the moment the system enforces (m_i, k_i) by exploiting the correctness of successful executions of task instances, while complying with (m_i, k_i). In the worst case, when all unreliable task versions are wrong, the system will stick to the pattern and will thus execute S-RE in the sense that the instances marked with "1" will be executed using full protection. However, all instances that are marked with "0" in the (m, k)-pattern will be executed with the error the detection version τ^d in order to detect errors or verify correctness and take further actions from there on.

This part of the thesis will first discuss the main principle behind postponing using examples and then prove why the enforcement of (m_i, k_i) can be postponed. After that, an algorithm will be presented which illustrates dynamic compensation in pseudo-code. Afterwards, at the end of this chapter examples will show the difference between the static and the dynamic approaches. This chapter concludes the basics which are necessary to understand the principles and techniques which are used in the experiments.

Proposition 1: Successful executions using τ_i^d postpone the enforcement of (m_i, k_i).

To prove this proposition we suppose that a static pattern Φ_i is given. As explained earlier, Φ_i is a binary string, containing the information about which version of a task should be executed. It was discussed earlier that Dynamic Compensation allows to exploit the correctness of completed and successful executions of unreliable instances. To be specific, Dynamic Compensation counts the successful executions of unreliable instances, and then utilized their correct execution. These completed and correct task instances are defined as S and stand for success. S allows us to greedily postpone the adoption of the original static pattern Φ_i. One can imagine that S is inserted into the bitstring, but only at the beginning of the pattern. S or multiple S can only be inserted before a zero, as only zeros give the system "a chance" to tolerate an error. If we take the bitstring $\{0, 1, 0, 1, 1\}$ for example, it will have the form $\{S, 0, 1, 0, 1\}$ at $t = 1$ and $\{S, S, 0, 1, 0\}$ after two time units $t = 2$ of a cycle, when $t = 0$ is the starting point. Because of the successful execution of two task instances which are denoted as S in the bitstring, the string at $t = 2$ now lacks the last two "1"s from the original bitstring. These two are pushed out of the string, and thus the adoption of the original pattern can be postponed. If the (m_i, k_i) pattern was $(3, 5)$, two instances out of $m_i = 3$ were already processed correctly at $t = 2$, which

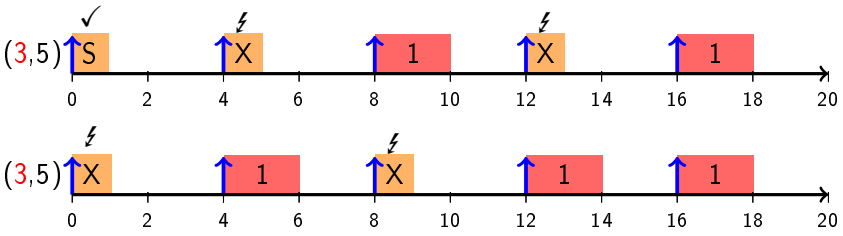

Figure 3.2: Example for the inserting S for $(3, 5)$ with the pattern $\{0, 1, 0, 1, 1\}$,adapted from [9]

means that only one more task instance needs to be correct for (m_i, k_i) to be fulfilled. It should be noted that S instances are always executed with τ_i^d in order to identify successful execution or errors. S can be explained as "giving a try with the detection version and having success with it".

In Figure 3.2 the insertion mechanism can be seen. In the upper diagram, the system gives a try with a detection version first, since the string begins with a "0". An S is inserted before the bitstring, because no error is detected. This means that we only need to execute two more task instances with using a reliable task version, since one was correct already, thus the adoption of the pattern is postponed and a "1" is pushed out at the end of the pattern. For the next instance, the system gives a try with the detection version again, but this time an error occurs. There was only one "0" in front of the "1", which means there is only one chance to fail. Because the chance was used, the system needs to run the reliable version in the third instance again. It gives a chance with the detection version on the fourth instance, but an error occurs, thus the pattern is not postponed any further and no more "1" are pushed out. In the lower diagram an error occurs in every instance of the detection version. This is the worst case, there is no postponing and no "1"s are pushed out, the systen just follows the pattern.

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 when inserted as S into the sequence of operations, task τ_i can still enforce the (m_i, k_i) constraint with the given pattern Φ_i for any consecutive k_i jobs, in which $x \geq 0$ [8].

The theorem was proven by using contradiction, assuming that the insertion of x successful executions of S violate (m_i, k_i) in the interval t to $t + k_i T_i$. For the contradiction to be wrong, the number of successful executions needs to be less than m_i within that interval, after inserting S. The interval starts with a 0,1 or S, including k_i consecutive executions. If x successful executions of S are inserted into the original sequence, then x original instances from the bitstring are pushed out. Successful executions can only be inserted before a 0 in the bitstring, because only zeros give a the chance to tolerate errors. Because S are only inserted before 0, only x "1"s can be pushed out. Therefore, for every inserted S, a 1 is pushed out, and because S is a correct execution, the amount of reliable instances is still m_i, thus the number of correct instances doesn't change when inserting

S. The contradiction is reached here. However, there are still questions that could arise: Were we just lucky that there are "1"s at the end of our bitsting ? What happens if the bitstring ends with a "0" ? Can we push "0"s out ?

To answer this issue, replenishment counters are introduced next, which separate the original pattern into sub-patterns which begin with 0 and end with 1. These counters monitor the current status of fault tolerance and aid in run time execution in the algorithm. If the pattern $\{0, 0, 1, 0, 1, 1\}$ is given, it is divided into $\{0, 0, 1\}$ and $\{0, 1, 1\}$, two partitions. The general rule is to divide the original static pattern into smaller patterns that start with 0 and end with 1. After rearranging, the number of partitions are counted and their number is saved in p_i. The counters o_{ij} and a_{ij} are needed next. The index of the task is denoted as i and j describes the current partition in which the system is working. Counter o_{ij} describes the number of unreliable instances in each partition and a_{ij} counts the number of reliable instances in the static pattern, but for each partition respectively. Consequently, the counter o_{ij} stands for the chances in a partition the system has to be wrong, as it records the number of the "0"s.

To give an example we consider the pattern $\{0, 0, 1, 0, 1, 1\}$. In this case $p_i = 2$, $o_{i1} = 2$, $a_{i1} = 1$, $o_{i2} = 1$, $a_{i2} = 2$, and thus $O_i = \{2, 1\}$ and $A_i = \{1, 2\}$. There are two unreliable instances in the first partition, $o_{i1} = 2$ and one reliable $a_{i1} = 1$, whereas in the second partition, there is one unreliable $o_{i2} = 1$ instance and there are two reliable instances $a_{i2} = 2$.

If o_{ij} is used up by errors, the system then has to follow the original pattern like in RE. To model that kind of switching behavior, a mode indicator Π is used to be able to distinguish the execution status of dynamic compensation. With the definition of $\Pi = \{tolerant, safe\}$ the current status of the task can be specified. If the tolerance counter o_{ij} is depleted and the system thus isn't allowed to give any more tries, because more insertions would violate the (m_i, k_i) constraint in case of an error. Π will then be set to safe. This causes the system to just execute the task instances with the reliable version. However, if the task can still tolerate errors because the tolerance counter is not depleted, then Π will be set to tolerant. In this case the system still has chances, and S could potentially be inserted into the pattern, thus the system is allowed to give a try with the detection version.

3.2.1 Algorithm for Dynamic Compensation

The algorithm works on one of the partitions of an (m, k)-pattern, at the beginning the index j will be passed while the mode will be set depending on whether the tolerance counter o_{ij} is not zero. Subsequently, if Π is in tolerant mode, the algorithm executes τ_i^d and saves whether a fault was detected or not. If a fault is detected, then the tolerance counter o_{ij} will be decreased by one, and after k instances it has to be increased back.

```
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
```

Algorithms 3.1: Dynamic compensation of task τ_i with (m_i, k_i), adapted from [8]

If o_{ij} is fully depleted, or to be exact equal to zero, then the system has to be set into safe mode which forces the system to only execute reliable instances. As explained earlier, a_{ij} stores the number of reliable executions, thus in Line 9 the value of a_{ij} is stored in l. If p_i is in safe mode, then l will be decremented step by step till it is equal to 0, when this happens the task will be set back to tolerant mode and the index j will be incremented, meaning that the next partition will be processed by the algorithm. When the system is in safe mode, there are two different strategies which can be pursued. Detection and Recovery (RE) will always execute the unreliable instance of the task with fault detection first, although o_{ij} is already depleted. Thus the system still gives a try with the detection version and only executes the reliable version if it finds an error in the detection version. Reliable Execution (RE) on the other hand will always execute the reliable version directly if the system is in safe mode. The techniques will be called D-DR (Dynamic Detection and Recovery) and D-RE (Dynamic Reliable Execution) in the rest of the thesis.

Because of Theorem 1 we know that the algorithm will always enforce (m_i, k_i). Successful executions of unreliable instances only postpone the enforcement of (m_i, k_i) and

in the worst case, if all unreliable instances are erroneous, the system will still follow the static pattern since no S were inserted.

3.3 Summary of Soft-Error Handling techniques

In this thesis five soft-error handling techniques were presented in total. They are ordered by their overall utilization, beginning with the highest value. D-DR shows the lowest overall utilization out of all proposed techniques [8]. To give a quick overview, the techniques are presented one below the other.

To give a quick overview, the proposed techniques in [8] are all shown in Figure 3.3, in which D-DR performs the lowest regarding overall utilization among them.

In Figure 3.3 in the first diagram the system uses S-RE and just follows the pattern $\{0, 1, 1\}$ for the constraint $(2, 3)$. When using S-DR, the system first gives a try with an unreliable version, but has to execute the correction version because faults occur in the second and third task instance. When D-RE is used, the system gives a try with the detection version in the first task instance. The task turns out to be correct, so the system still has one chance to be wrong, thus it gives a try with the detection version again in the second task instance. An error occurs in the second instance, but since the system can tolerate one error due to its constraint, it can move on without taking action against the error. However, the system has to execute a reliable task version in the third task instance to comply to $(2, 3)$, since an error already occurred in the second instance. The only difference between D-DR and D-RE is that D-DR still gives a try with the detection version, although the constraint would be broken if an error occurred. The system detects an error and executes the reliable version afterwards.

- Fully Robust (FR): Runs all instances with reliable version τ_i^r.
- Pattern-Based Execution (default task version: τ_i^u)
  - S-RE: Runs task version τ_i^r for instances marked as "1".
  - S-DR: Runs task version τ_i^d for instances marked as "1" and recovers executing τ_i^r if an error is detected in τ_i^d.
- Dynamic Compensation (default task version: τ_i^d)
  - D-RE: Runs execution version τ_i^r for instances marked as "1" if the tolerance counter is depleted.
  - D-DR: Runs execution version τ_i^d for instances marked as "1" if the tolerance counter is depleted. Runs execution version τ_i^r for recovery if an error is detected in τ_i^d.

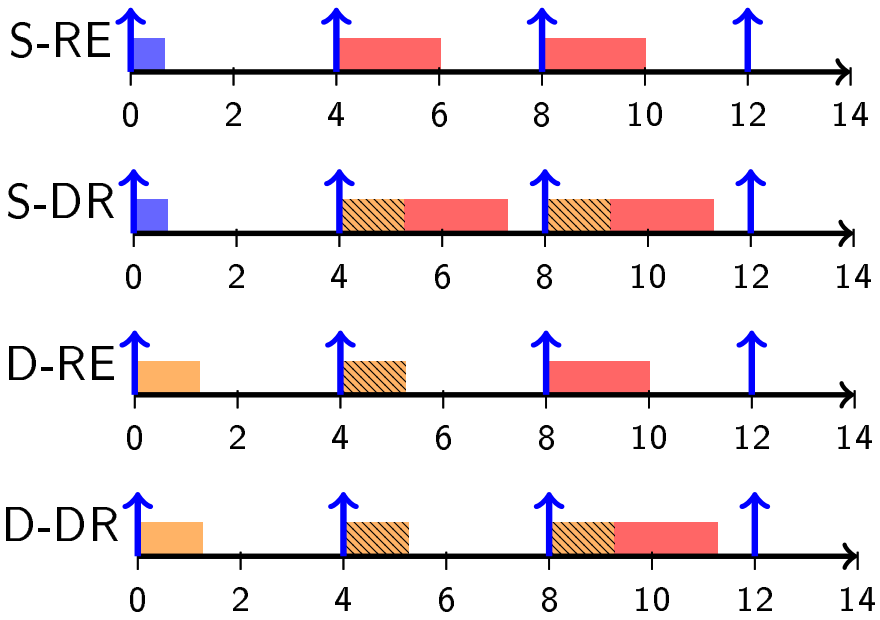

Figure 3.3: This example illustrates the different techniques. (m_i, k_i) is $(2, 3)$ and the static pattern is $\Phi = (0, 1, 1)$. Soft-errors happen in the second and third instance and are marked with stripes. The Blue block is unreliable, the orange block is the version with detection, and red block is reliable. Adapted from [8].

Chapter 4

Hard- and Software used for Experiments

In this chapter, the hard- and software used in the experiments will be described. The chapter will begin with the presentation of the robot hardware and its parts into which faults are injected. After that, the Real-Time Operating System nxtOSEK will be presented briefly.

4.1 Lego Mindstorms NXT Robot

The Lego Mindstorms NXT brick will be used as the computing unit for all experiments in this thesis. It runs the Real-Time Operating System nxtOSEK which allows us to code custom applications with C/C++ and provides access to the robots sensors and motors.

Figure 4.1: The Lego Minstorms NXT brick. All pictures of photos of the robot are from shop.lego.com

The brick can take input from its ports at the bottom, which can connect up to fours sensors. On the top are three output ports located which can control up to three motors. The self-balancing robot [22] uses two motors and four sensors are connected, of which only three will be used in the experiments. On the bottom are two light sensors located, which are able to help to follow a black line, which is drawn on paper. If there is no enough light, the sensors can turn on their infrared lights on to be able to track the line.

Figure 4.2: NXT Servo Motor

Figure 4.3: NXT Light Sensor

The microcontroller is powered by an Atmel 32-bit ARM main processor, which runs at 48 Mhz, while having 64 KB RAM and 256 KB flash memory. The co-processor is an 8-bit AVR ATmega48 processor running at 8 Mhz with 512 Bytes RAM and 4 KB flash [23, p. 12].

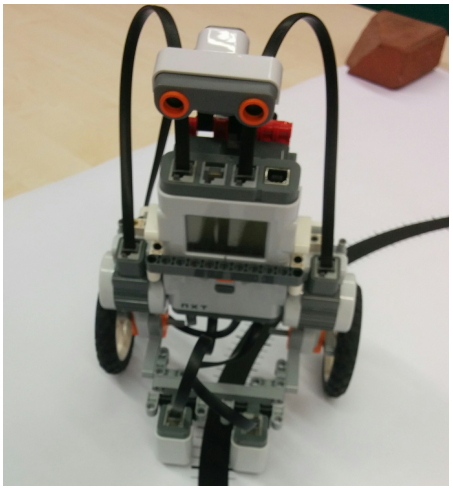

Figure 4.4: The self-balancing robot on a circular track, from [9]

Figure 4.5: Gyro Sensor

Figure 4.4 shows the complete robot. Its core consists of the brick, and the two servo motors can be seen on the sides. The two wheels are directly connected to the motors, whereas the motors are connected to the brick through the ports on top. On the very bottom, the two light sensors are mounted and connected to the brick through the ports at the bottom. By keeping the line between the sensors, the robot can trace the path. A sonar sensor is installed in top the brick, but is not used for experiments in this thesis. The Gyro Sensor which can be seen in Figure 4.5 is mounted behind the brick, behind the red Lego-piece over the USB-port.

4.2 nxtOSEK

The open source platform OSEK was developed for and by several different companies from all around the world in the automotive industry. It offers the possibility of implementing real-time operating systems on hardware in the automotive industry. The port of OSEK on the NXT provides a C/C++ programming environment on the robot using the GCC tool chain. The C API allows us to access the motor, sensors, the display, and other devices. It also allows us to upload programs in C directly to the memory.

Applications which run on nxtOSEK mainly have to consist of two types of files. The OIL file has to be written in the OIL language. With it one can declare tasks and their alarms, stacksize, etc. which provides the freedom to modify low level operations. In the C file, motor-, display-, sensor-and other functions can be accessed. However, tasks defined in the OIL file have to be called before calling these funtions. Examples about implementation details will be covered in the following chapters. More information can be found on the homepage of the project [24].

4.3 OSEK Implementation Language (OIL)

The inforation for this section can be looked up in [25]. As explained earlier, applications for this robot consist of two parts, there is the OIL-file and the C-file. Tasks are defined in the OIL-file, while tasks are called in the C-file. In the OIL file, a set of OIL objects can be defined, while the CPU is the container for all objects. OS, TASK, COUNTER, ALARM, and RESOURCE are just a few OIL objects, and the ones used in this application. The file in our application starts with the CPU container with the CPU specification. At the end of the curly bracket, a semicolon has to be included.

```
CPU ATMEL_AT91SAM7S256
{
        ...
};
```

Code 4.1: The CPU container.

To initialize the operating system, hook routines and OS status need to be defined. We define STATUS as EXTENDED, while no hook routines are used.

```
CPU ATMEL_AT91SAM7S256
{
  OS LEJOS_OSEK
  {
    STATUS = EXTENDED;
    STARTUPHOOK = FALSE;
    SHUTDOWNHOOK = FALSE;
```

```
    PRETASKHOOK = FALSE;
    POSTTASKHOOK = FALSE;
    USEGETSERVICEID = FALSE;
    USEPARAMETERACCESS = FALSE;
    USERESSCHEDULER = FALSE;
  };

  APPMODE appmode1{};

  ...
};
```

Code 4.2: Initialising OSEK.

Next, APPMODE needs to be set, which defines the different modes of operation for an application. For nxtOSEK applications, appmode1 always has to be used. Afterwards, task objects can be defined. AUTOSTART decides whether a task is activated at system start-up. If FALSE is used, an ALARM-object needs to be defined to activate a defined task. With PRIORITY the tasks priority can be set. The lowest priority is zero (0) and larger numbers denote a lower priority. With ACTIVATION the maximum number of queued activation requests for a task can be set. When it is set to "1", then it means that only once instance of a task can be active at any single time. With SCHUDULE, the preemptability of a task can be set. FULL means preemtable and NON means non-preemptable. STACKISIZE specifies the size of the stack of a task.

```
CPU ATMEL_AT91SAM7S256
{
  OS LEJOS_OSEK
  {
        ...
  };
  /* Definitions of a periodical task: OSEK_Task_ts1

  TASK OSEK_Task_ts1
  {
    AUTOSTART = FALSE;
    PRIORITY = 2;
    ACTIVATION = 1;
    SCHEDULE = FULL;
    STACKSIZE = 512; /* bytes */
  };
```

```
  ALARM OSEK_Alarm_task_ts1
  {
    COUNTER = SysTimerCnt;
    ACTION = ACTIVATETASK
    {
      TASK = OSEK_Task_ts1;
    };
    AUTOSTART = TRUE
    {
      APPMODE = appmode1;
      ALARMTIME = 1;
      CYCLETIME = 4;
    };
  };

  ...

  COUNTER SysTimerCnt
  {
    MINCYCLE = 1;
    MAXALLOWEDVALUE = 10000;
    TICKSPERBASE = 1;
  };
};
```

Code 4.3: Defining tasks, alarms, and system counters in OSEK

To activate the task, an alarm needs to be set. The alarm works on basis of a counter, which is defined down below. The action of the alarm is to activate the task using ACTIVATE-TASK and the task name need to specified. The alarm should be started automatically, while appmode needs to be specified the same as earlier. ALARMTIME should always be set to "1", meaning it will expire after one time unit. CYCLETIME activates the alarm every four time units. The counter represents the tick source for alarms. A tick is a time unit used for measuring internal time, the smallest unit of time which can be distunguished by the system. TICKSPERBASE sets the number of ticks which are needed to reach the counter-specific unit. MAXALLOWEDVALE limits the counter value, in our case the counter value is not allowed to surpass 10000. For a cyclic alarm which is linked to the counter, MINCYCLE specifies the minimum allowed number of counter ticks, here it is set to "1". With these tools new tasks can be implemented or the existing ones can be modified. For futher information on OIL, the official document about System Generation by the OSEK group can be referred to [25].

4.4 Calling Tasks in C

The tasks defined in the OIL file can be called in the C file and their specifications will be applied. Tasks need to be terminated at the end, to interact correctly with alarms and new task instances of the same task due to the periodicity provided by nxtOSEK.

```
TASK(OSEK_Task_ts1)
{
        ...

        TerminateTask();
}

...
TASK(OSEK_Task_ts2)
{
        ...

        TerminateTask();
}
```

Code 4.4: Calling tasks in C

Chapter 5

The Studied Application

In the following sections, the inner workings and underlying concepts of the different tasks will be discussed. It is important to understand how the application works, to comprehend the effects of faults on the application performance. In the subsequent part, the main topic will be how and where faults were injected in previous experiments in [8], as well as where faults will be injected in further experiments.

The application mainly consists of three tasks, the first task is responsible for keeping the robot balanced on two wheels. The second task traces the path using the light sensors, while a third task runs in the background and is responsible for showing output on the display. A fuzzy approach, which is used in the path tracing and balancing task, allows the robot to move more fluidly, while balancing out outliers by taking the weight off of single computation results through considering past computation results in the decision making process.

5.1 Fuzzy Table

Fuzzy technology can be used to enhance fault tolerance on control systems. The path tracing application for example first reads crisp sensor values, and fuzzifies them. In the next step it computes an output value based on the last ten input values and proceeds with defuzzifying. This produces crisp values, which can be used by the system again. A fuzzy approach enhances the fault tolerance in this system, because the influence of single faults on the control application is decreased through the consideration of outputs of past computations. Using a fuzzy table also smoothens the values, and balances outliers. Thus, if wrong sensor or motor data occur, its effect will be counterbalanced by using the fuzzy table, and as a consequence the application can tolerate faulty values. Secondly, the path tracing and the balancing of the robot operate smoother, e.g. the steering behavior of the robot becomes less abrupt when the fuzzy table makes the value of the the steering angle change slowly. The configurations of the fuzzy table, e.g. its size, the threshold at which a one is stored in the fuzzy table for path tracing, the way of summation of the motor input

values, and the weighting of the current and last ten motor input values, all these setting are chosen on empirical basis. The system could be run with other settings, but testing them out is out of scope for this work.

5.2 Error Handling

In this application only the path tracing and the balancing task adopt soft-error handling techniques for fault tolerance. Depending on the system setting, both tasks are executed with Static or Dynamic Compensation. After the balancing task and the path tracer task are activated by their alarms, the system has to decide which task version the task at hand has to execute. When the system executes this the path tracing task, three versions as discussed in Section 2.4.3 are available: unreliable version, fault detection version and reliable version. Depending on the system setting, it uses either Static Compensation or Dynamic Compensation to decide the task version. Algorithm 3.1 or a simpler version to execute a static technique will executed. This is the "top layer" of this application, consequently the task version is chosen before any other activity starts. It is not difficult to notice that protection techniques cause overhead. Before the application starts, Algorithm 3.1 will be executed, taking up resources, execution time and memory for counters and patterns. For the simplicity of presentation, the path racing task is chosen in the following example to explain the how the compensation techniques work.

5.3 Path Tracing

The path tracing task allows the robot to trace a black line on a track. After choosing the task version, the system will first call the function which gets the data from the light sensors. To avoid abrupt steering maneuvers, a fuzzy table is used which smoothens the steering values of the robot. The values from the light sensors are read first and then fuzzyfied to covert them into a fuzzy format to store them in the fuzzy table; the information from the sensor is converted into "1" or "0". These values are then stored in a table which stores ten entries, the ten last light sensor values. The table has two rows and ten columns. In the first row, the "1"s and "0"s for the decision "turn right" are stored, whereas in the second row the information for "turn left" is stored. In every instance of a task, the system will read light sensor values from the left and the right sensor once and will store the corresponding information in the fuzzy table. Processing the light sensor values is illustrated in Figure 5.1 in the first to states in the blue state. To write sensor information

rightsum	1	0	1	1	0	1	0	0	1	1	$\sum = 6$
leftsum	0	0	0	1	1	1	1	0	0	1	$\sum = 5$

Table 5.1: The fuzzy table for the turn value

on form of 1 or 0 into the fuzzy table, sensor data has to be read first. Depending on whether the sensor data exceeds a certain threshold or not, 1 or 0 are written into the fuzzy table, while simultaneously adapting the index of the fuzzy structure. In practice, that means that if the light sensor is detecting a certain lack of light, then it means that this sensor has to be hovering right above the black (lightless) path line. Imagine the robot tracing the path, but with the light sensors just besides the black line. Assuming we stand behind the robot and it drives away from us on a straight line, if the right light sensor suddenly hovers over the line, then the robot has to turn to the direction of the light sensor which detected less light, e.g. if the right sensor detects that it is above the path line, then the robot has to turn right, otherwise it would leave the track, since the left sensor will be outside the track if the path is not too wide.

The threshold which dictates whether 1 or 0 is written into the fuzzy table, is 600 in this case, the maximum possible value from the sensor is 1023 and the minimum value is 0. A greater value means less light (or lower reflection). After storing the 1 or 0 from the right sensor into the first row and the 1 or 0 from the left sensor into the second row, the index pointer is incremented, rerolling it by working with the modulo operator if the index exceeds the length of the table. That way the oldest value will always be overwritten. This is the process of fuzzifying sensor data and storing it into the table. After that the "1"s need to be summed up to calculate a value called "turn" which is the information needed for steering.

The turn value defined as a static variable at the very beginning of the code, so it will keep its value between invocations. It tells the robot to turn to a certain direction and is modified by dedicated function. At first, there is a loop which iterates through all ten entries of this table. The values rightsum and leftsum which are defined at the beginning of the function serve as a means to add up the entries in the fuzzy table. As explained earlier, the entries in the fuzzytable save the ten recent decisions to turn the robot to the left and to the right in form on one and zero.

The turn value is the actual steering value, and the extreme values are defined as -100 (turn sharply to the left) and +100 (turn sharply to the right). First, leftsum is calculated and subtracted from the turn value, next, rightsum is calculated and its calue is added to the turn value afterwards. If the subtraction of leftsum and rightsum result to zero, then it means that they had the same value, thus no steering takes place. If leftsum is higher than rightsum, then more will be subtracted from the turn value, which corresponds to making a turn to the left. If the number of ones for calculating rightsum are higher, the turn value will be positive, which will cause a turn to the right.

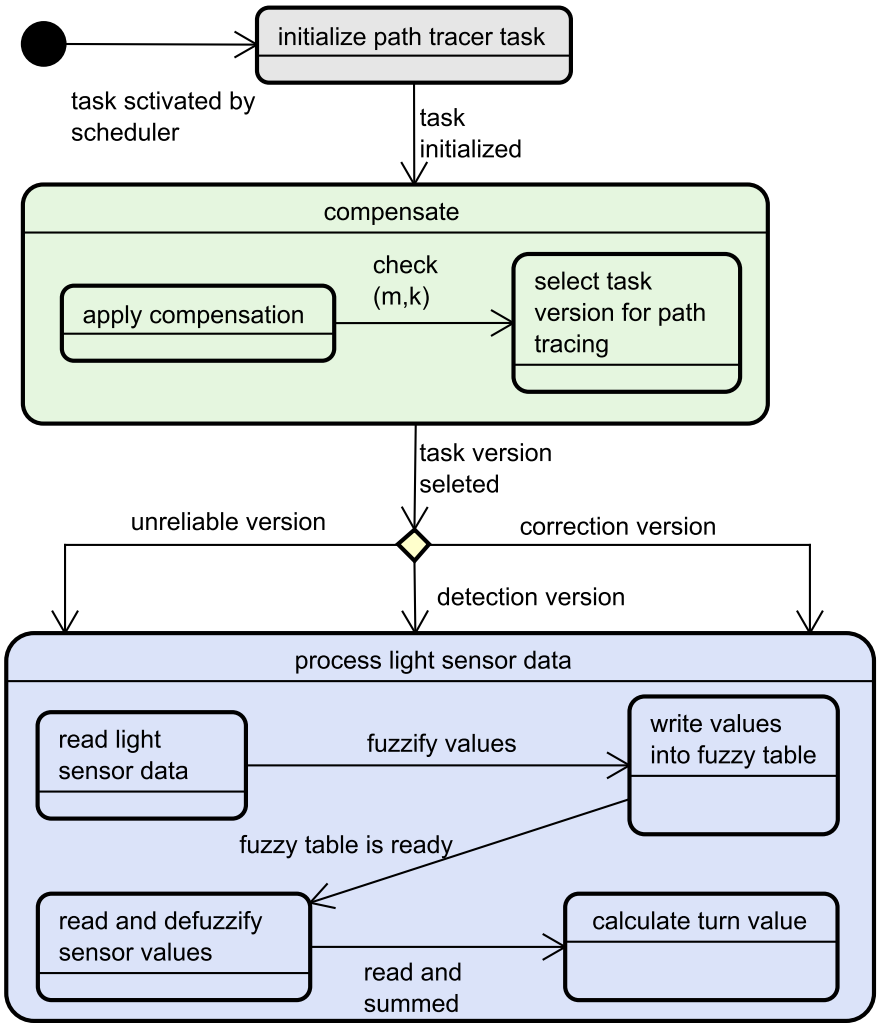

Figure 5.1: The path tracer task illustrated in a flowchart

- If leftsum= 0, nothing will be subtracted from the turn value.
- If $1 \leq$ leftsum ≤ 3 after summing up the ten recent entries, it means that there were two or three "1"s in the last ten light sensor samplings. In this case 50 will be subtracted from the turn value.
- In case of $4 \leq$ leftsum ≤ 6, then 80 will be subtracted.
- In case of leftsum≥ 7, then 100 will be subtracted.

The value rightsum is calculated the same way, the only difference is that the values are added and not subtracted. In the end the corresponding values of rightsum and leftsum are added to each other. In Table 5.1, the resulting turn value is $-80 + 80 = 0$, since rightsum is six and leftsum is five, thus the robot will drive forward without turning. If

leftsum is equal to zero and rightsum equal to ten, then the turn value is +100, this causes the robot to turn to the right.

Figure 5.1 illustrates the the path tracing task in a flowcart. To summarize, the path tracing task calls the functions which start the compensation, which in turn call the path tracing code, while deciding the execution version of a task. When the path tracing task is starts, it samples sensor data from the light sensors first. Then the sampled data is fuzzyfied to ones and zeros and stored in the fuzzy table. After filling the fuzzy table with the newest light sensor sampling data, the ten values which correspond to previous sensor sampling data, the ones and zeros in the fuzzy table are summed up, to calculate the current turn value. The application then leaves the turn value, till the balance task finishes processing it.

5.4 NXTway-GS (Self-Balancing)

The first and most important duty this task has is to balance the robot on two wheels. After initializing the task and its values, the application calls the balance control function. The internal control theory will be treated like a black box in this thesis. Faults are not injected into the balance controller, but on the output value of the balance control. The balance control code [26] was developed using MATLAB with Simulink and was generated automatically, and a detailed description is out of scope for this work, only the properties which are manipulated in this work will be discussed. For this work, the most important arguments are power motor left (pwml) and power motor right (pwmr). They are defined at the beginning, are signed 8 bit values and determine the strength of the impulses which are sent to the motors. Although they appear in the parameter list, these are the output values of the balance function and thus cause the wheels to turn in a specific manner, including turning and forward motion, while balancing the robot on two wheels. At the end of the task these two values are sent directly to left and right motor respectively.

After the balance function finishes executing, there are three execution versions for sending the motor input value to the motors, the unreliable-, fault detection-, and error detection version. The fuzzy table works differently in this task. After trying out different settings, the conclusion was drawn that fuzzifying the motor input into "1" and "0" or to 10, 20, 30,... etc. is not possible, since the robot can't balance itself in that case. The task's motor input values change very slowly, and tend to oscillate around a certain values with small changes, as can be seen in Table 5.2. This behaviour needs to be preserved for the balancer to work properly. This is the reason for using the real values, seperating them in groups destroy the balancer. The last ten motor input values are saved in the fuzzy table, but for the left and right motor separately. The first step in the application is always to store the current motor steering value (which is not sent to the motor yet) in the fuzzy table. Subsequently, all the last ten entries in the fuzzy table are added and then divided by ten, to calculate he average of the last ten motor input values of the last ten

rightsum	-4	-4	-11	-9	-8	-8	-7	-6	-6	-5
leftsum	-4	-4	-11	-9	-8	-8	-7	-6	-6	-5

Table 5.2: The fuzzy table for the motor input values. In this example the values are the ten first values after the 16. balancer task instance.

task instances. Then, in the last step, the average of the last ten values is added to the motor input value of the current instance, and then halved. Weighting them equally, this calculates their average, and this value is sent to the motors, determining their impulses. Formula 5.1 shows the formula to calculate the motor input value (miv). First all ten fuzzy table values (ftv) are summed up, then added to the current miv (cmiv), and in the end the sum is divided by two. To sum up the way of working of the balancer task, Figue 5.2 shows that the balance function is executed first, followed by the compensation. Then the motor input value is read and stored in the fuzzy table. Formula 5.1 calculates the current motor input value, which then is sent to the motors.

$$miv = \frac{\frac{(\sum ftv)}{10} + cmiv}{2} \tag{5.1}$$

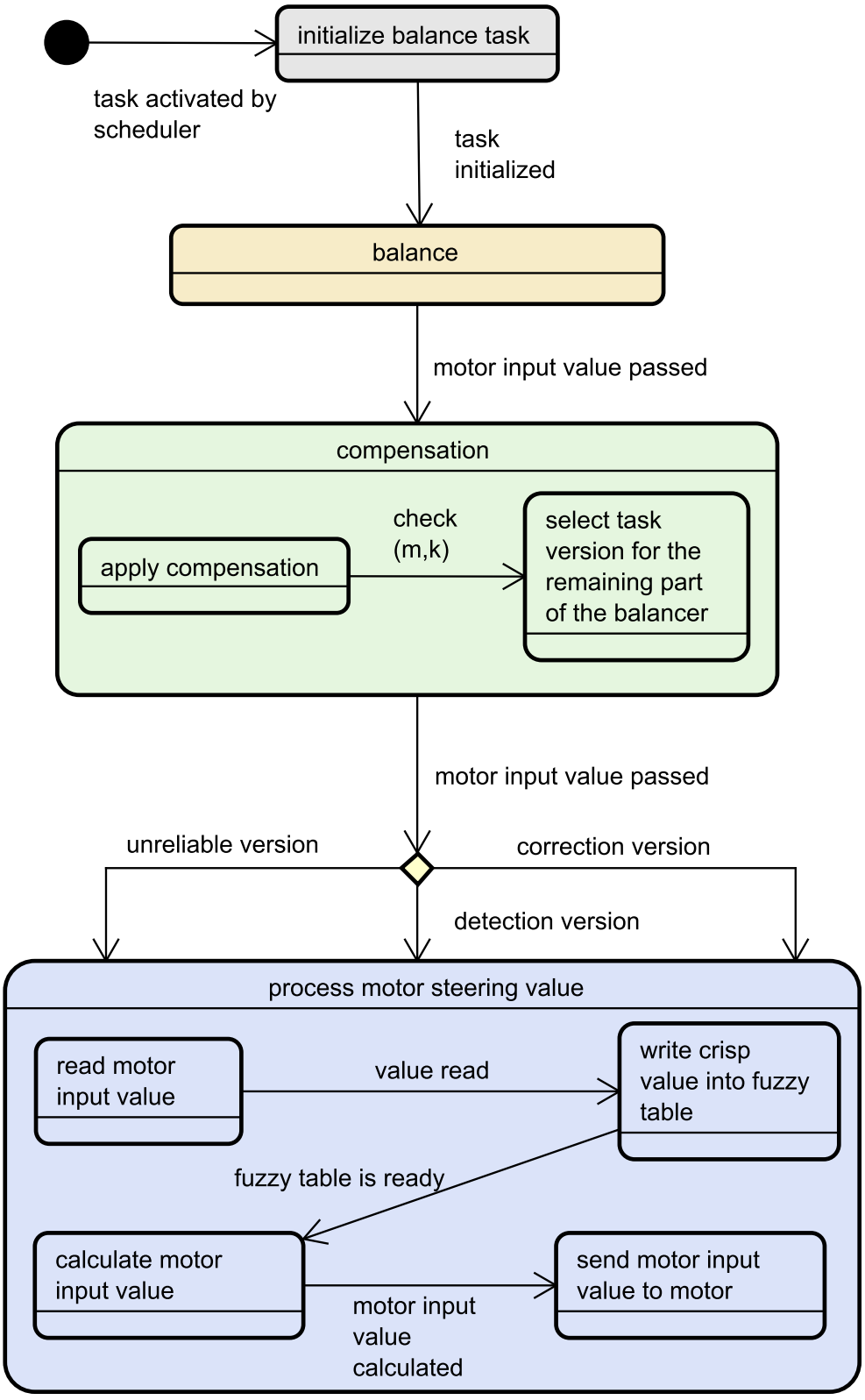

Figure 5.2: The balancer task illustrated in a flowchart

Chapter 6

Fault Injection

This chapter will cover how and where faults are injected in the experiments. First, the original design of fault injection into sensor sampling data in [8] is discussed, then the fault injection into motor steering values is explained. To avoid confusion, it is important to note that every instance of an unreliable task only has certain values which are prone to faults. In case of the path tracing task, only the light sensor value is prone to faults. In the same way, faults are only injected into the output value of the balance function. The τ_i^u version of the task is always prone to faults, even when it is used to execute τ_i^d, when two identical task instances are needed to be able to compare, these two identical task instances are unreliable τ_i^u instances. In the same way, the three task instances which are needed to correct errors when executing τ_i^r, are also prone faults since every single one of them is an unreliable τ_i^u instance. Generally speaking, only τ_i^u are available, and τ_i^d and τ_i^r are built using τ_i^u as a basis and then comparing them to each other.

6.1 Decisions to Inject Faults

To inject faults, an experiment function is declared in a separate file. This function has an object as a parameter, which stores certain information about the injection, e.g. the fault rate, and the number of injections. In the beginning of this experiment function, a random number is generated, so a "dice" is rolled, by calling a random number generating function. After incrementing the counter for the number of injections, the dice-value is compared to the a value which is generated depending on the fault rate of the specific task. If the the fault rate is small, then it is more likely for the random number to be bigger, so fault injection is less likely. Depending on the result of the comparison, the binary is returned to the calling function. The experiment function is only called by the path tracing and the balancer task, to inform them about whether a fault should be injected. The two tasks always store their fault rates and pass them to the experiment function.

```
U8 experiment(unrel_exp_t* param)
{
        int dice = rand();
        ++param->value_tot;
        if (dice > param->value_rate)
        {
                return FI_SUCCESS; // no fault
        }
        ++param->value_cnt; // increment counter for number of injections
        return FI_FAIL; // inject fault
}
```

Code 6.1: The experiment function

6.2 Fault Injection in Light Sensor Values

When executing the path tracing task, the function responsible for compensation and thus deciding which task version to execute depending on the compensation setting is called. All three task versions do the same execution steps. They read the sensor data from the light sensors, then convert the data to store it in the fuzzy table, and subsequently evaluate the data in the table to calculate the turn value, which decides the steering action. Fault injections in this task takes place after the original light sensor value has been read, as can be seen by following the arrows to the red state in Figure 6.1. The task first calls an injection function, which passes the fault rate to the experiment function, thus obtaining the information whether a fault should be injected. Then, this information (inject or don't inject a fault) in the injection function makes it either return the correct or a faulty value, depending on the injection information. When the current task is an unreliable instance without detection or correction, then the value which is returned when a fault needs to be injected is either 0x0 or 0xFFFF, the minimum and maximum possible light sensor values, and the decision is made by comparing the correct value to 0x7FFF, which is 0xFFFF divided by two. If the correct value is lower 0x0 will be returned, and if the correct value is higher 0xFFFF will be returned to the calling function, which is the path tracing task.

If the current task version is an error-detecting one, then an error detecting injection function will be called instead, in which the unreliable version of the inject-function will be called two times. To have a means to compare two values and detect a fault, the same sensor value needs to be computed twice. This means that the experiment function and thus the "dice-rolling" will be will be absolved twice, since each sampling of an unreliable sensor value is prone to faults.

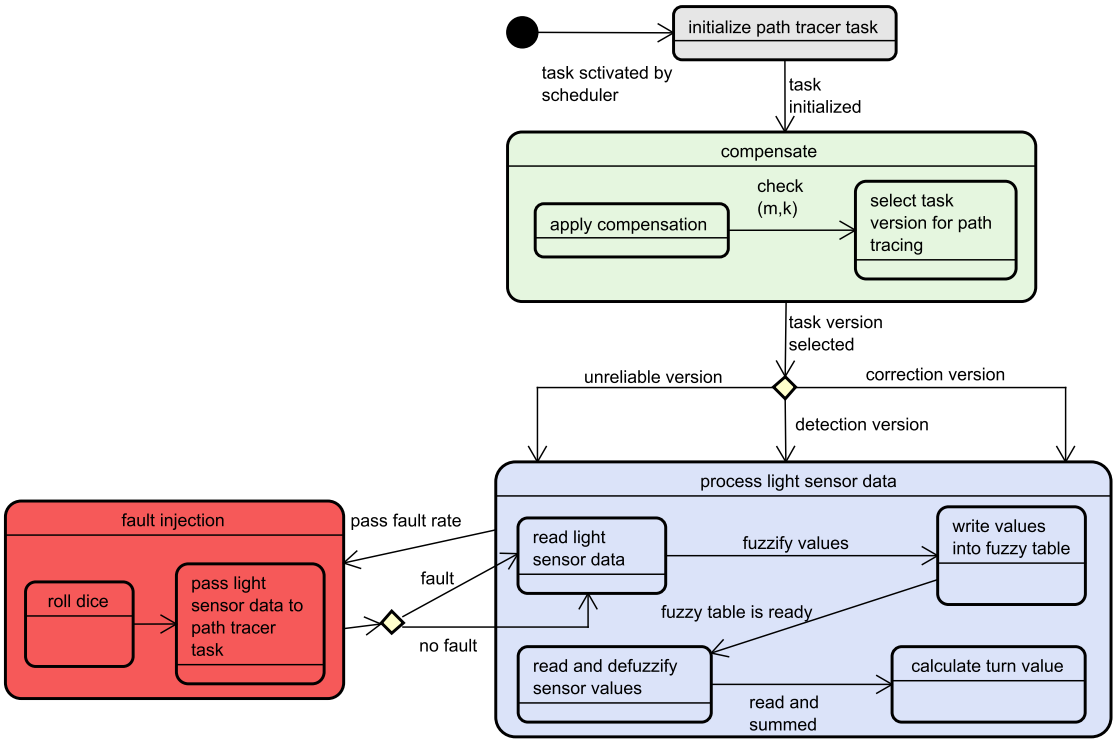

Figure 6.1: The path tracing task illustrated in a flowchart, with fault injection

6.3 Fault Injection in Motor Steering Values

At first, the task calls the balance control function, which is responsible for balancing the robot. As explained earlier, the balancer function has two output values, pwml (power motor left) and pwmr (power motor right). The faults are injected into these two values, but only after they are generated by the balance control function, this is highlighted in Figure 6.2, compensation and fault injection only take place after the balance function executes (yellow state).

Earlier experiments concluded that the balance application is very sensitive towards faults. Even very low fault rates cause the control loop to crash and it can only function using the constraint $(1, 1)$. Due to the complexity of the balancer application and its vulnerability to faults, the injection takes place on the two output values of the balancer function, which are the motor impulses for the left and right motor forward motion and steering. These two output values are computed and then send to the motors. The balance control model indicates that these two values are not sent back in form of feedback to the control function. However, experiments support the thesis that the injection of faults into these values can still destroy the control loop and make the robot lose balance and fall. These two output values are used for steering and forward speed, but not for balancing, thus fault injection in this task will take place after the values leave the balance function.

When the motor input values are computed, they are ready to be send to the motors. Just like in the path tracing task, the compensation function is called, which decides the execution version of the task instance, but in this case the motor input value has to be passed down as well. First, the tasks sends the fault rate and the correct motor input values to the injection function, which in turn calls the experiment function to decide whether a fault should be injected, and then returns a faulty value in case of fault, and the correct value in case of no fault. The value which is returned to the balancer task is either the minimum or the maximum allowed motor input value, which is $+100$ or -100. When the balancer task receives the value, it stores it in the fuzzy table. Subsequently, the task evaluated the fuzzy table and calculates a motor input value using the last ten entries, while weighting the current value and the fuzzy value equally as in Formula 5.1. After this, the task sends the motor input value to the corresponding motor. It is important to note that faults are injected independently in the left and in the right motor value.

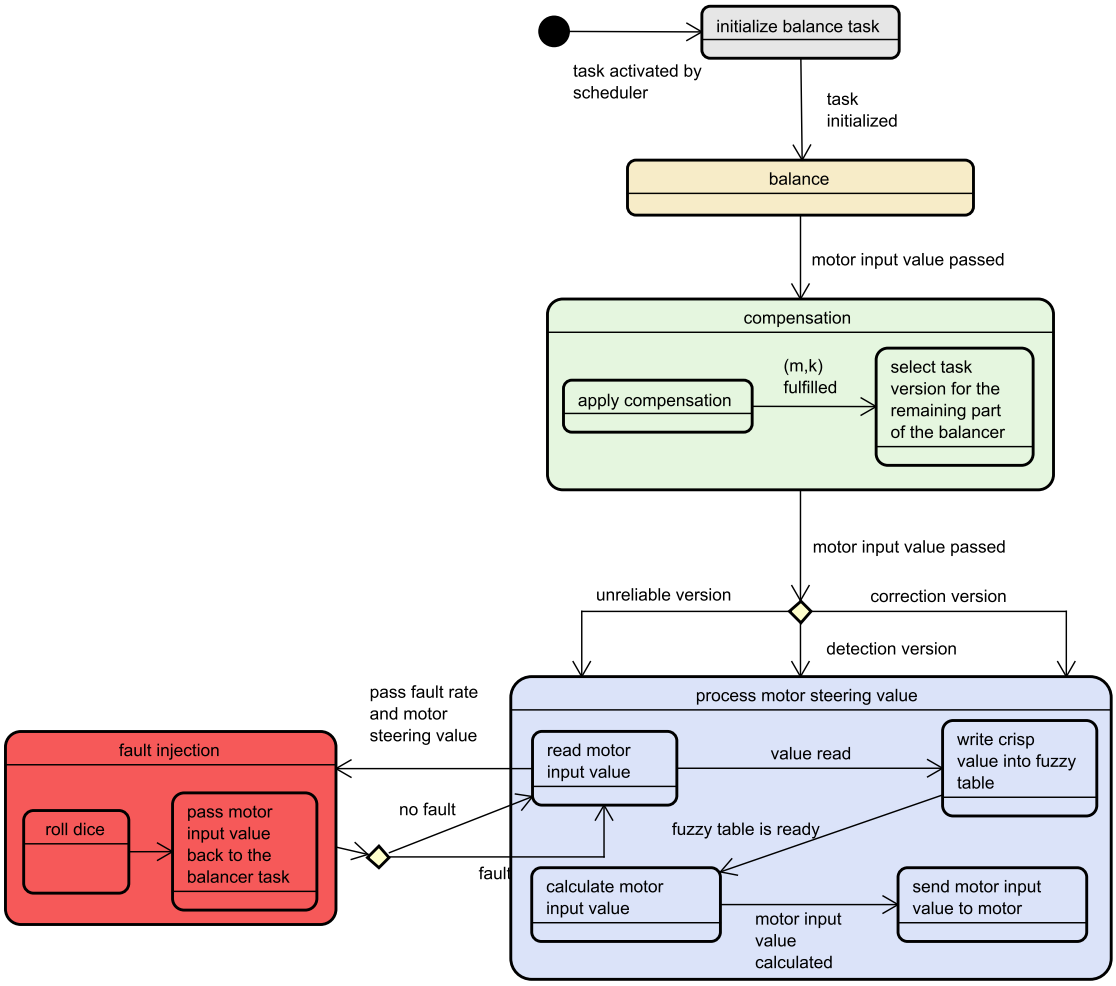

Figure 6.2: The balancer task illustrated in a flowchart, with fault injection

Chapter 7

Fault Injection Experiments

As explained in Section 1.2, one of the main goals of this thesis are to find (m, k) for the path tracing and the balancing task through experiments, and afterwards testing these constraint by enforcing them with a certain fault rate to find out if the constraint prevents the experiment from failing. If enforcing the (m, k) constraint succeeds in keeping the robot on the track till the very end of it, then we empirically prove that (m, k) exists and can be enforced to guarantee successful runs, thus preventing failures. In this chapter, the experiment setup will be explained first. Then, the empirical method to find possible (m, k) candidates will be discussed, and the experiment results from the search for (m, k) for sensor and motor injection will be presented. The subsequent section will present the experiment data for (m, k)-enforcement with different experiment settings.

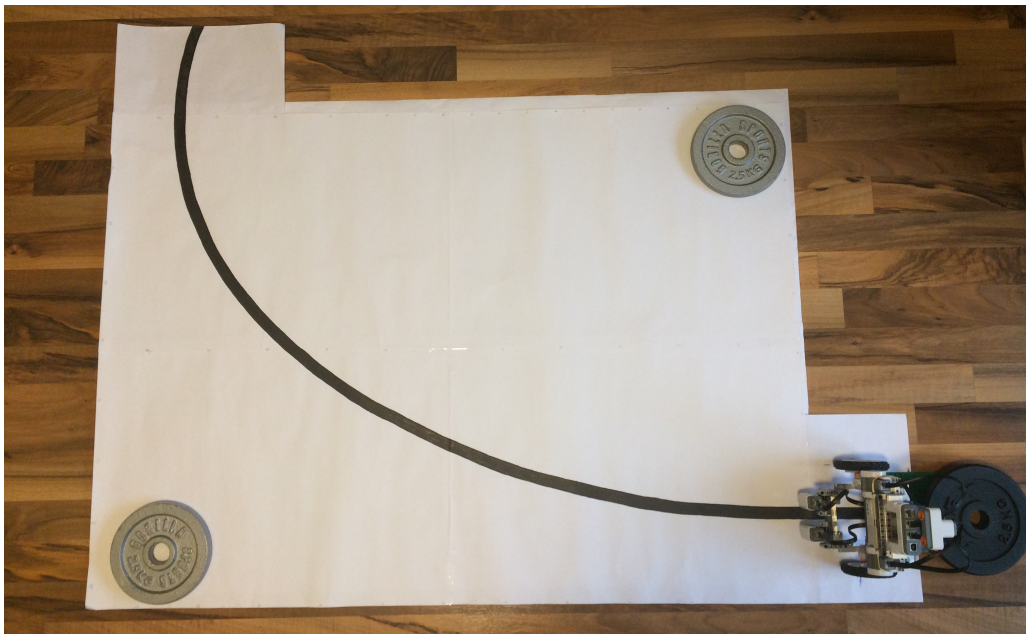

Figure 7.1: The whole track from above

7.1 Experiment Setup

The size of the paper the track is drawn on is Din A0 $(84,1 \times 118,9cm)$. When measured till the inner line, the elliptical track can be described with $a = 105$ cm and $b = 68$ cm, a width of 2 cm, and a a total lengths of around 160 cm. The reason for the choice of this specific track is, inter alia, that it features elements of a straight and a curved line. The robot will always begin on the side with the smaller radius unless specified otherwise. The first 30-40 cm proceed almost like a straight line, and the curvature increases the more the robot reaches the end of the track.

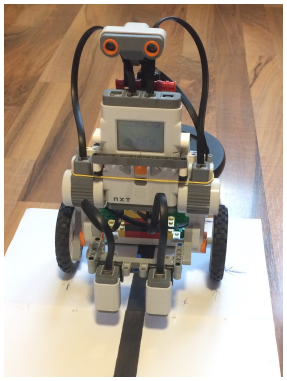

Figure 7.2: Initial position of the robot

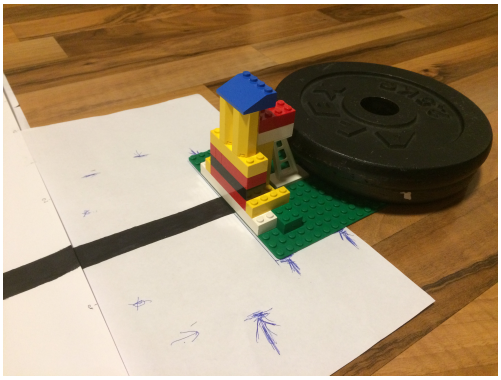

Figure 7.3: Starting block

In [27] the simulation settings "straight line" and "curved line" are chosen for their specific experiments in which the path tracking control of four-wheel steering autonomous with vehicles with actuator failues is examined. An ellipse is a combination of the two. Because tests were already done on a circular track for sensor injection in [1], a different track was chosen to prove that (m, k) also exists for different types of tracks. An elliptical track is also mathematically easy to describe, easy to reproduce, and measuring distances can be done precisely in a reasonable manner. The distances will be measured at the end of the robot through measuring the angles. In the area around $\frac{\pi}{2}$, at the end of the track, the ellipse can be approximated as circle with a constant radius, which simplifies comparing distances of different experiment runs, because the reached distance before or after $\frac{\pi}{2}$ can be calculated by the angle alone. This is also the reason why there is a piece of paper which extends the end of the track. The starting point also has a paper attached to it, because the space is needed, so that the robots sensors begin on the main part of the track. The paper also ensures that the ground profile and the light distribution is always consistent. The starting block ensures that the robot starts in a balances state. It has a height of around 8.5 cm and was built using simple Lego stones. It is always places 10.5 cm afar from the main paper.

A bit of practice is needed to familiarize with the initial starting angle of the robot when it is put against the starting block to begin the experiment. If the robot leans too much forwards when leaning against the block, it would fall without the balance control.

It will try to compensate that imbalance by driving forward very fast for a few centimeters. The same applies to the scenario in which the robot is tilted backwards too much, then it might stand still for a few moments, trying to push itself against the starting block, to reach a balanced states. If this movement in the beginning is too severe, it can affect the experiment results, so when setting the robot up by leaning it against the starting block, the initial angle should roughly be always the same, without any deviation ideally. This issue caused a bit of confusion when doing the experiments, but this will be explained in the following sections.

There are also two more tracks available, they only differ in width. The first track has a width of 2 cm, whereas the thinner line has a width of 1 cm and the thicker one a width of 4.5 cm. 2 cm is roughly the distance between the two light sensors, while 1 is smaller than this distance, and 4.5 cm exceeds the distance between the light sensors, so the robot registers black on both sensors in the starting position. The course of action is finding (m, k) using the 2 cm track and then trying to apply this (m, k) to the thinner and thicker tracks to find out if it still guarantees runs without. The prediction is that a thinner line needs a higher constraint, whereas a thicker line manages with a looser constraint.

7.2 Finding (m, k) empirically

Because the robot's I/O limitations, e.g. lack of a file system, no internet, problematic bluetooth connectivity, (m, k) candidates need to be displayed on the built-in screen, and the process of analyzing experiment data needs be done on the robot's hardware. To analyze experiment data and to be able to find (m, k), the experiment part of the application features a data structure to record the number of correct and faulty task instances, called experiment set. The memory units in this hardware consists of bytes, and the type of the executed task version is stored in one continuous bitmap. A task with a fault is stored as a "0" whereas fault free tasks are stored as a "1". Whenever a fault is injected, the currently active task calls a function which adds a "0" to the experiment set, standing for "the current task is erroneous". If no fault is injected and the running task is correct, a "1" is added to the bitmap. This way all executed task versions can be recorded, which allows us to analyze the experiment set, and thus find possible (m, k) candidates.

The experiment set has a char pointer which stores the starting memory address of the experiment set. There is also a pointer to the memory address of the last byte of the memory for the set, which can be set in the beginning to determine the experiment set size. A third char pointer to a memory address provides an index to store the information about the current position in the bitmap. A second index provides information about the current position in a byte cell. Figure 7.4 illustrates the bitmap with its pointers and its index for the bitposition. If we assume that the string 10111101...00000101...00000000 is the experiment set which is being filled, a possible constellation could be that the first pointer stores the memory location of the first byte, the second pointer stores the address

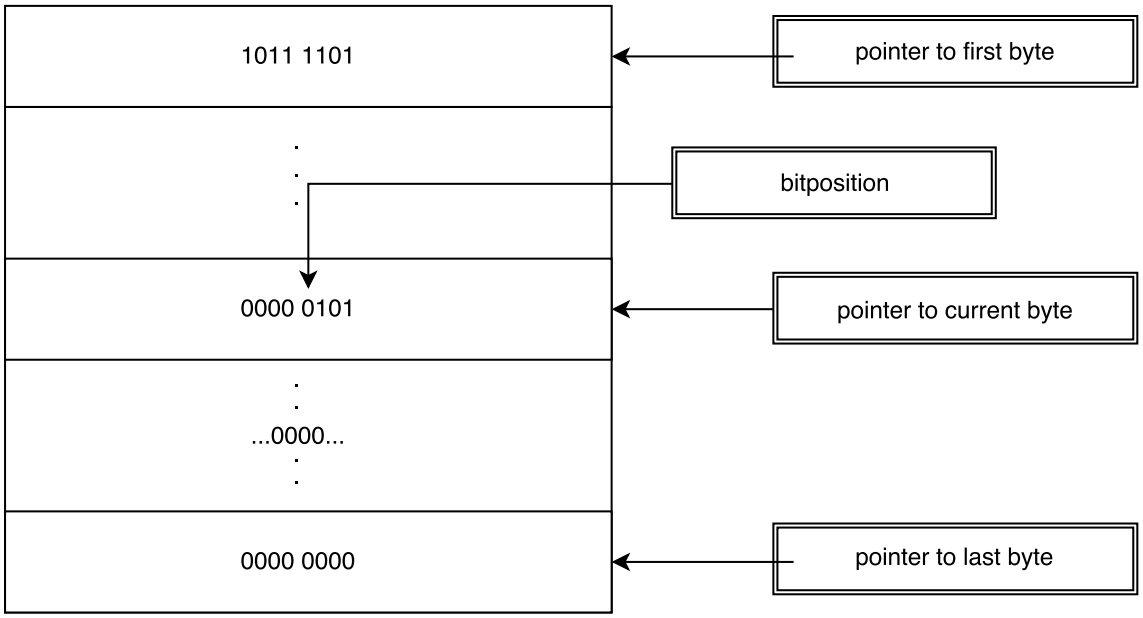

Figure 7.4: The structure of the bitmap in the memory

of the current byte which is filled at the moment, the third pointer determines the end of the bitmap, while an unsigned 8-bit integer index provides information about the current bit up to which the buffer is filled, e.g. it could be the seventh bit in the second byte when counting from the right. "1"s and "0"s are added to the set by shifting the bit in the integer variable to the left, and then using AND or OR operations to insert it into the set. It is rerolled if the highest bit is reached, while the pointer which points at the byte which is currently filled is incremented, which causes a jump to the next byte, in case the one byte is full. When setting the size of the experiment set, the number of bytes is used. In the experiments which were done for this thesis, a experiment set size of 664 and 560 bytes was chosen, since the size of the sets decides the total number of executed tasks, and a number around 600 lets the robot drive till the very end of the track. This number needs to be divisible by eight, since the byte cells are addressed in 8-bit steps.

The (m, k) constraint can then be found by letting the robot trace the path successfully first, while recording the "1"s and "0"s in the experiment set. When the robot finishes tracing, a background task starts analyzing the experiment set, and tries to find minimal m for different window sizes. At first, the loop checks the minimal m e.g. beginning with $k = 3$ in a sliding window, this means that the first three bits of the experiment set are checked, and the minimal number of "1"s is stored. Then, the second, third, and fourth bits are analyzed, and again the minimal number of "1"s is stored. After the analysis of $(m, 3)$ is finished, the minimal m is displayed on the screen of the robot. Subsequently, the same procedure is done with $(m, 4)$, and m is increased by one after the analysis finishes, and this goes up to $k = 16$.

Depending on the fault injection rate, the displayed (m, k) will differ, m will be lower naturally the higher the fault rate, since there will be more faulty tasks. The goal is to find minimal m, and this requires pushing the fault rate near the maximum. The fact that there were a minimum of m correct task instances in a sliding window size k, the experiment didn't fail, and this is the reason why the recorded (m, k) should in theory prevent the robot from failing when enforced on system with fault injection.

Only if the run is successful then the possible (m, k) are noted down. If the fault rate gets too high, the robot will get off track, or lose balance and fall down, counting as a failed run, and the (m, k) of the failed runs are never noted down.

7.2.1 (m, k) Candidates for Sensor Injection

When trying to find (m, k) candidates, a version of the system without any protection or compensation is used. We are only interested in finding a minimum of m correct instances in a given window size k, with which the experiment succeeds. For this reason faults were injected with different fault rates. Beginning with a fault rate of 5% per instance, the displayed (m, k) were noted down. Increasing the fault rate slowly in 5% steps gives enough time to get used to the experiment, especially setting the robot in a neutral angle at the very beginning. The size of the experiment set was chosen as 664 bytes, as this setting gives us the desired length of the experiment, while the forward speed will always be 25 in all experiments, unless specified otherwise.

When the fault rate is low, between 5% and 15%, then the recorded (m, k) differ a lot. $(12, 16)$, $(10, 16)$, and $(7, 16)$ can be found for these fault rates. These should only be chosen for enforcement if the quality of control of the task with below15% is desired. However, this much quality of control is not needed, as there are less strict (m, k) which also keep the robot on track, so choosing these constraints would be over-provisioning. The faults were almost unnoticeable, and did not affect the robots ability to trace the path, so the fault rate needed to be increased further. From 20% to 30% (m, k) seems to stabilize a bit with $(5, 16)$ in both cases.

Then, from 30% fault rate on, there seems to be a sharp line, $(5, 20)$ appears all the time up to around 52% fault rate. It appeared in every try when experiments were made in steps of 1%, so for 35 up to 52% all tries displayed $(5, 20)$, and thus a stable (m, k) constraint was found. Another alternative constraint, $(4, 16)$ appeared almost all the time too, but ultimately $(5, 20)$ was chosen, as it easier to enforce by the compensation.

If the fault rate is over around 52%, then the experiment fails, the robot doesn't trace the path sufficiently and leaves the track. For the reason, the (m, k) constraints from failed runs are not useful for our purposes.

Fault rate %	5	10	15	20	25	30	35	40
(m,k)	(12,16)	(10,16)	(7,16)	(5,16)	(5,16)	(5,20)	(5,20)	(5,20)

Figure 7.5: (m, k) constraints and fault rates for sensor injection, excerpt from Table A.1 and Table A.2

7.2.2 (m, k) Candidates for Motor Injection

In this part, fault injection into sensor data was completely turned off, as we are only interested in the behavior of the system when faults are injected into the actuators. Finding an (m, k) constraint was not as straightforward as in sensor injection. When running experiments with increasing fault rates, e.g. 30%, 60%, and 100%, the balance of the robot was not severely disrupted as it was assumed. The higher the fault rate, the more the robot swung back and forth, but never losing its balance completely. Thus, the robot could still trace the path successfully without leaving it, only the distance decreased because the jolting causes the robot to stop or drive backwards for a few millimeter between the forward leaps. The behavior of the robot with 100% fault rate can be compared to the the shaking of a car with manual transmission, when a high gear is engaged during its start-up, but it's shaking during the entirety of the run. This situation is not suitable for injecting faults and searching (m, k) candidates, since no clear distinction can be made between a failed and a successful run. It would be much more desirable to have the robot lose balance and fall, or leave the track when injecting faults into the motors.

The reason for the fact that the robot doesn't lose balance despite a fault rate of 100% is that we use a fuzzy table to smoothen erroneous input. The values which are put into the fuzzy table when injecting errors are $+100$ and -100. If we assume that $+100$ and -100 appear equally often, then it could be the case that the faults cancel each other out in the fuzzy table, thus passing a rather neutral value to the motors. To gauge the flow of the motor input value without fault injection, the motor input values were displayed on the screen in earlier experiments, the values changed slowly, with no abrupt changes, and they oscillated from positive to negative during the measurements.

To further test this assumption, only $+100$ is used as the value which is used to inject an error. This gives us the behavior we want, the robot actually loses balance and falls down with a very low fault rate. A fault rate around 16% is enough to cause the robot to lose balance, and this is the maximum fault rate for this task. $(10, 16)$ was chosen as the candidate, as it is more stable and allows more quality of control, and it corresponds to a pattern of tasks with around 12% fault rate.

If faults are injected with a higher percentage than 16%, then the balancing of the robot behaves like a bridge which is stimulated by external forces which match its natural oscillation, leading to its destruction trough intense swaying. Since the robot drives on two wheels and balances itself through weight transfer in an oscillating manner, we can

assume that the system has its own natural oscillation. Mechanical resonance occurs when a system's response to a stimulation with its natural frequency is a greater amplitude of movement. When too many faults are injected, the robot makes a leap forward, and has to balance itself by driving backwards again, then has to balance out that backward movement by leaping forward again. This way it increases its own swaying every time it tries to correct the previous one, and thus falls on its back or on its sensors, losing balance. It can be concluded that although the injection takes place outside of the balancer application, the motor input value still has a major impact on the control loop and is able to corrupt it, thus it makes sense to protect it from soft-errors.

Fault rate %	6	7	8	9	10	12	14	16
(m,k)	(11,16)	(11,16)	(11,16)	(11,16)	(11,16)	(10,16)	(10,16)	(8,16)

Table 7.1: (m, k) constraints and fault rates for motor injection, Table A.3 and Table A.4

7.3 Testing (m, k)

In the following sections the constraints $(5, 20)$ for the sensors and $(10, 16)$ for the motors is tested on a system with error compensation, with varying fault rates. The fault rates are always chosen significantly above the maximum tolerable fault rate of the tasks, so that the system has to compensate and enforce (m, k) to prevent the system from failing. If the fault rate would be below the maximum fault rate, then the compensation would not prevent the system from failing, thus a fault rate needs to be chosen which undoubtedly leads to a fail.

To prove empirically that a combination of a (m, k) constraint with a specific fault rate keeps the robot from failing, all combinations of constraints and fault rates are conducted with 20 runs. When testing 25% with $(10, 16)$ for motor injection for example, 20 runs of the experiment with these settings are done. If all 20 runs are successful, then the combination of the constraint and the fault rate is marked as "Y". If only a few, e.g. one or two runs failed, then the experiment set and the starting angle of the robot is checked. If these sources of errors can be excluded, then the combination of the constraint and the fault rate is marked as "N", and we assume that the experiment fails due to a loose constraint. If the number of failed runs is significantly higher than one or two, then the combination is also marked as "N" too. This is done for all combinations in Table 7.2 and Table 7.3.

The compensation technique used to enforce (m, k) in the following sections will be Static Reliable Execution. The focus of these experiments is testing whether the previously found (m, k) prevent the system from failing, and for these purposes S-RE suffices, while keeping the workload of the compensation low. Moreover, the difference in performance

between e.g. S-RE and D-DR in this system is not noticeable in the experiment with the naked eye, and is only noticeable when measuring the execution time of the tasks.

7.3.1 Testing (m, k) for Sensor Injection

At first, $(5, 20)$ with a fault rate of 60% was tried. The robot did not track the path at all using S-RE, and the experiment failed every time. The feeling came up that changing (m, k) doesn't make a difference in the system. Changing $(5, 20)$ to $(20, 20)$ in the application still showed the same results, no tracing, and there was no difference in the robot's behavior. $(20, 20)$ did not show the behavior which is expected from a constraint which guarantees "every instance correct", it just behaved like a system without protection. It seemed like no compensation was taking place.

After checking out the code, it becomes clear where the issue lies for this problem. When correcting faults, unreliable faulty instances are used. If there is a 60% fault rate, more than half of the instances will be faulty, and this leaves the error correction version vulnerable to faults. If these faulty values are used for correcting, and compared to each other in majority voting, then the result of the majority voting and thus the "reliable value" will be incorrect. For this issue there can be four possible scenarios:

- If all three values are correct when no fault was injected into either, then the output of the majority voting will be correct.
- Two correct values and one faulty one is the second scenario, in this case the output will also be correct.
- If there are two faulty values and one correct value, then the output of the majority voting will be incorrect, and the reliable instance will not be reliable any more.
- When all three instances are incorrect, the output will we incorrect too. With a 60% fault rate the first and second scenario are not very likely, but therefore the third and fourth are more likely to happen. Considering the fact that the injection uses the same minimum or maximum values for an erroneous value, it is likely that there are three different values in the majority voting.

To remedy this issue, "ideal correction" is used, meaning that the error correction instances will always be correct, no matter how high the fault rate. This is done by simply replacing the code in the error detection version with a direct call to the light sensor sampling function, removing the injection part completely from this version of the task. This makes error correction instances independent from unreliable instances, and allows us to enforce (m, k). Error correction versions will theoretically always be correct, barring real soft-errors.

After replacing the code in the reliable version of the path tracer with "ideal correction", with the standard light sensor sampling function in the API, $(5, 20)$ was tested again with

60% fault rate. With this setting the path tracing is successful. To gauge how low m can get, it was reduced step by step, and it turns out $(2, 20)$ is the constraint with the lowest m, which keeps the robot on track. The same experiment was done for 80% fault rate, and it turns out that m needs to be higher when the fault rate is higher.

(m,k)	(5,20)	(4,20)	(3,20)	(2,20)	(1,20)
60% FR	Y	Y	Y	Y	N
80% FR	Y	N	N	N	N
100% FR	Y	N	N	N	N

Table 7.2: (m, k) constraints for sensor injection with fault rates of 60%, 80%, and 100% where "Y" stands for stable and "N" stands for fail - number of tries can be looked up in Table A.1

7.3.2 Testing (m, k) for Motor Injection

To gauge the stability of $(10, 16)$, three different fault rates are chosen. 25%, 50%, and 100% are all higher than the maximum tolerable fault rate of the balance task, and cover low-, mid-, and high fault rate scenarios.

It turns out that $(10, 16)$, our initial candidate, is never stable, and always leads to a loss of balance, even with only 25% fault rate. There is not one constraint that is stable for all fault rates, except for $(16, 16)$. The higher the fault rate, the higher m needs to be for the system to not fail. However, the lower m, the more the robot tends to swing back and forth, but can balance itself again when m is high enough. The fails are all due to the intense swinging of the robot, it either falls to the front on its two light sensors, or on its back on its gyro sensor. The lower m and the higher the fault rate, the more intense the swaying. One could argue that 20% fault rate should be chosen too, and $(10, 16)$ could be enough to keep the robot in balance, but 20% is too near to the maximum fault rate. There is not a strict boundary for the maximum fault rate, but it is lower than 20%, and that is too close to $16 - 17\%$, the effects of the compensation might not take any effect, and the robot might balance itself without compensation, so fault rates of which is known to definitely cause a fault need to be chosen.

(m,k)	(16,16)	(15,16)	(14,16)	(13,16)	(12,16)	(11,16)	(10,16)
25% FR	Y	Y	Y	Y	Y	Y	N
50% FR	Y	Y	Y	N	N	N	N
100% FR	Y	N	N	N	N	N	N

Table 7.3: (m, k) constraints for motor injection with fault rates of 25%, 50%, and 100% where "Y" stands for stable and "N" stands for fail - - number of tries can be looked up in Table A.2

7.4 Case Study

7.4.1 Concurrent Fault Injection into Motors and Sensors

In reality, faults can occur in every part of the hardware. Generally, injecting faults by choosing the part of the hardware into which the fault should be injected randomly is an ideal representation of faults, but such injection methods would be very expensive. Allowing faults in the balance and path tracing task creates a model which is a little more realistic, since faults can occur in the motor input and in light sensor values. Enforcing $(5, 20)$ with a fault rate of 60% for the light sensors and $(11, 16)$ with 25% for the motors, the robot fails in 10% of all runs, by getting off track. Trying the same experiment setting with $(8, 20)$ and $(13, 16)$, the robot successfully traces the path in all runs, while making a better impression overall following the path more closely.

7.4.2 Different Track Widths

All experiments were done on a track with a widths of 2 cm. To find out whether the (m, k) constraints which were found in Section 7.2.1 still keep the robot from driving off track and provide a certain quality of control depending on the choice of the constraint, experiments with different track widths. In this section, only fault injection into the sensors will be studied, since injecting faults into the motors mainly disturbs the balance of the robot, causing no significant steering actions.

There are two additional tracks available. The thinner line has a widths of $1cm$, while the thicker line is 4.5 cm wide. The 1 cm track is thinner than the distance between the two light sensors, while the 4.5 cm line is so wide that the two light sensors are over the black line in the starting position. The idea is to have one line to be thinner, while the other line to be wider than the distance between the light sensors. The goal is to find out whether $(5, 20)$, which was found in Section 7.2.1, can still prevent the system from failing when a different path line widths is chosen. The assumptions are the following:

- To keep the quality of control high when a thinner path line is chosen, the system needs a tighter (m, k) constraint, since it has less leeway to recover from errors.
- If a thicker path line is chosen, then a more relaxed (m, k) constraint can be chosen to have sufficient quality of control, since it has more leeway to recover.

To prove the first point empirically, the $(5, 20)$ constraint is enforced on the line with 1 cm width, with a fault rate of 80% on the path tracing task. 80% is chosen in this case since this fault rate causes the system to fail every time without protection, while $m = 5$ is minimal, as can be seen in Table 7.2.

In the first set of the experiment, the robot only successfully traces the path in 7/20 tries. The robot always goes off track near the end of the track, when the curvature

increases. $(5, 20)$ does not suffice for a track with 1 cm widths, the system fails in more than half of the tries, and this is the reason why m has to be increased.

To increase m significantly with the goal to prevent the robot from having failed runs, the constraint $(8, 20)$ is chosen next. The rest of the settings remain the same. This time the robot traces the path better, with 18/20 successful tries. There are still unsuccessful tries, and to be able to guarantee a successful run, m has to be increased further.

$(10, 20)$ is chosen to weed out unsuccessful runs. Enforcing this constraint keeps the robot on the track in every run (20/20 successful tries), without unsuccessful runs. $(10, 20)$ seems to be the constraint which has to be chosen if a system is desired which never leaves the track.

For the second point, $(5, 20)$ is enforced first with 80% fault rate on a track with 4.5 cm widths. For this setting the tracing task makes a solid impression, and always traces the task without going off track. Testing $(4, 20)$ shows the same result, the robot traces the path perfectly without problems. However, trying $(3, 20)$ the robot never traces the task and tracing doesn't seem to work for such a low m. $(4, 20)$ is sufficient to prevent the robot from failing in this case.

7.4.3 Overall Utilization

In [8] the four error handling techniques S-RE, S-DR, D-RE, and D-DR are applied on a path tracing task with a $(3, 10)$ constraint, which was found and tested using the same methods which are described in this thesis. The reason why finding (m, k) is so important is the fact that it allows us to minimize energy consumption and overall utilization by applying these techniques. It can be seen in Figure 7.6 that the experiment results of the overall utilization test in [8] indicate that D-DR performs lowest regarding overall utilization, with varying fault rates and increasing m.

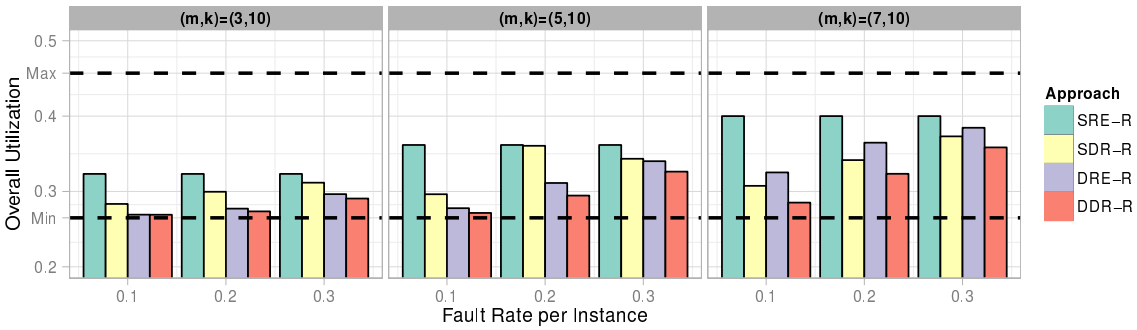

Figure 7.6: Overall utilization measured for the task path tracing with a $(3, 10)$ constraint [8]

We also found and tested (m, k) robustness requirements on path tracing with $(5, 20)$ and balance control tasks with $(11, 16)$ successfully in Sections 7.2 and 7.3. With those robustness requirements, we evaluate the effectiveness of the four compensation techniques, to prove that the (m, k) requirements we found can be used to reduce the overall utilization

(5,20), 20% FR	S-RE	S-DR	D-RE	D-DR
p	4741	4739	4744	4741
u	3365	3555	x	x
d	x	1184	4279	4276
r	1185	413	465	465
c_i^u	160	160	x	x
c_i^d	x	180	180	180
c_i^r	300	300	300	300
$\mathbb{U}$	0.19	0.19	0.19	0.19

Table 7.4: Values needed to calculate overall utilization

$\mathbb{U}$. To calculate the overall utilization, the execution times of each task version, the number of executions of each task version, the period of the task, and the number of all executions is needed. We denote the values u, d, and r as the number of unreliable, detection, and reliable versions of a task, respectively. The value p is the number of periods (e.g. the number of times the path tracing task is activated).

To calculate the overall utilization, we profile the execution time of each task versions: $c_i^u = 160$ μs, $c_i^d = 180$ μs, and $c_i^r = 300$ μs. The period of path tracing task is $T_i = 1000$ μs. In Table 7.4, the execution numbers for each task version and the number of periods are illustrated. The way we calculate the overall utilization are as follows:

$$\mathbb{U}_{SRE} = \frac{u \cdot c_i^u + r \cdot c_i^r}{p \cdot T_i} \tag{7.1}$$

$$\mathbb{U}_{SDR} = \frac{u \cdot c_i^u + c_i^d \cdot (d - r) + c_i^r \cdot (d + r)}{p \cdot T_i} \tag{7.2}$$

$$\mathbb{U}_{DRE} = \frac{c_i^d \cdot d + c_i^r \cdot r}{p \cdot T_i} \tag{7.3}$$

$$\mathbb{U}_{DDR} = \frac{c_i^d \cdot (d - r) + c_i^r \cdot (d + r)}{p \cdot T_i} \tag{7.4}$$

To cover the worst case, the highest recorded execution time should be used to calculate overall utilization. An example calculation follows:

$$\mathbb{U}_{SRE} = \frac{3365 \cdot 160 + 1185 \cdot 300}{1000 \cdot 4741} = 0.19 \tag{7.5}$$

It turns out that finding a setting for which the techniques show the argued benefit is not an easy task. In Table 7.4 for example, all values for overall utilization show no difference at all. The main reason for this problem lies in the length of the execution times. Since the execution time of unreliable, detection, and reliable versions were too close in our system, executing reliable versions was not punished with a sufficient cost.

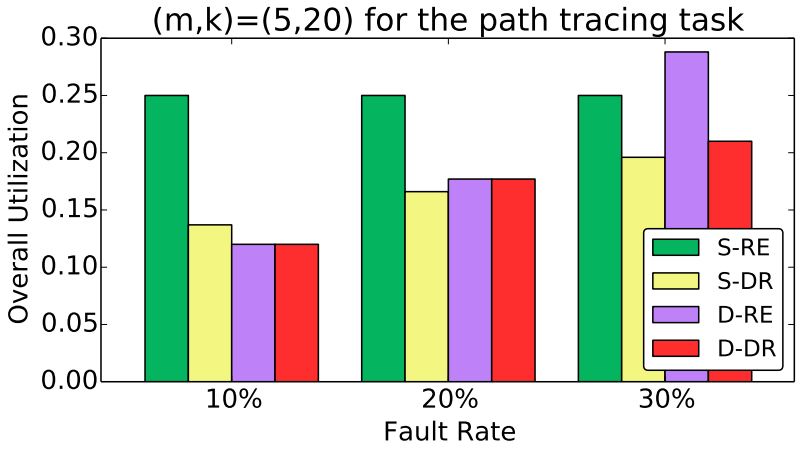

Figure 7.7: (5,20) with different fault rates for path tracing

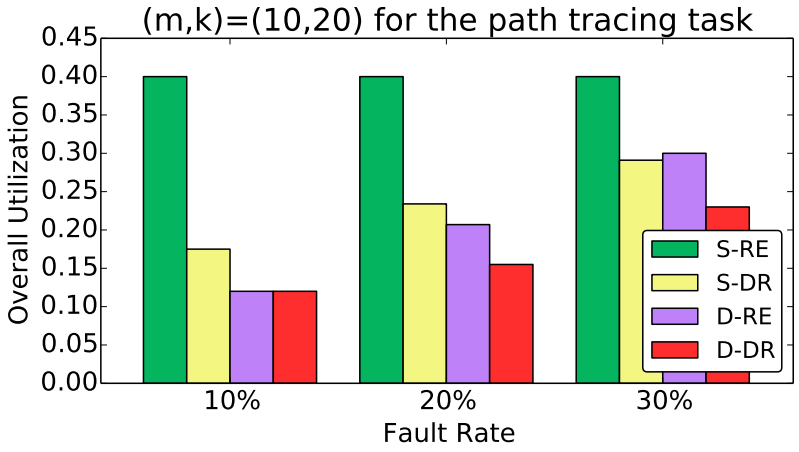

Figure 7.8: (10,20) with different fault rates for path tracing

In order to make the execution of reliable versions more costly, we changed the execution times of our reliable instances to 700 μs, while the detection version is set to 120 μs and the unreliable version to 100 μs, which are motivated by safety-critical systems. In the avionics industry for example, five identical instances are used as the multiple replicas to recover from soft-errors [28].

When calculating the results[1] for the $(5, 20)$ robustness requirement for the path tracing task, Figures 7.7-7.9 indicate that D-DR always outperforms S-RE in terms of overall utilization. When choosing tighter (m, k) requirements than $(5, 20)$, D-DR always outperforms all the other techniques, as can be seen in Figures 7.8-7.9. Interestingly, the results of S-DR are often close to the results of D-DR. Since S-DR also benefits from runtime decision, it can even outperform D-DR in case of $(5, 20)$. If we only consider the static compensation techniques, i.e, S-RE and S-DR, we can see that the increased m makes

[1] Data for calculating overall utilization can be found in Figure B.1.

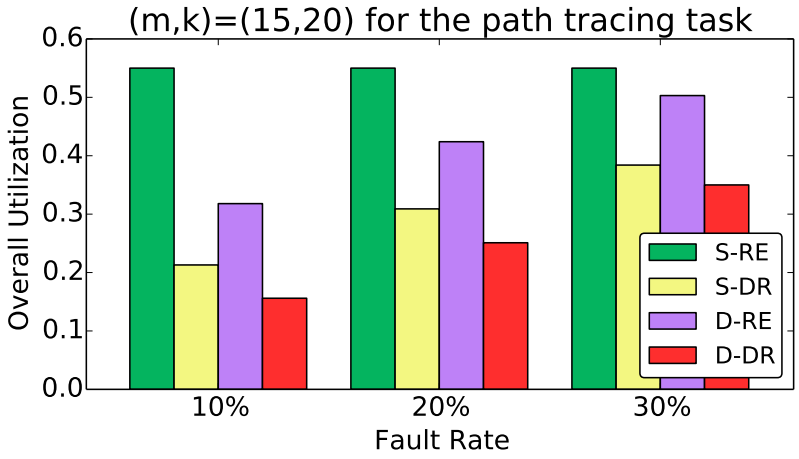

Figure 7.9: (15,20) with different fault rates for path tracing

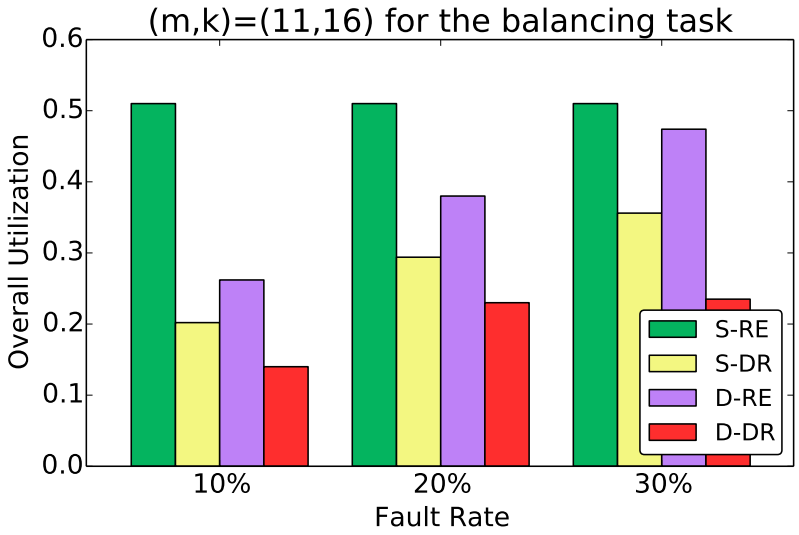

Figure 7.10: (11,16) with different fault rates for balance control

S-RE consume a lot utilization. However, S-DR can save a lot overall utilization, since mostly there are no faults in the additional tries with the detection version.

The more we tighten (m, k) requirements, the higher the difference between S-RE and D-DR. The biggest margin is 0.35 for $(15, 20)$ and 10% fault rate, confirming the effectiveness for D-DR with low fault rates. On the other hand, all experiments indicate that the margin between S-RE and D-DR decreases when the fault rates are higher. The most distinct example is in Figure 7.7 with 30% fault rate; the difference between S-RE and D-DR is strikingly low with 0.04. We can observe that the fault rate has more impact on the overall utilization ratio between D-DR and S-RE than the tightness of (m, k) robustness requirements. By exploring $(11, 16)$ for the balance control, we can observe that D-DR still outperforms all the other techniques under all tested fault rates. We only present $(11, 16)$, since it is already a tight requirement, and has a similar tightness to $(15, 20)$.

7.5 Analysis of Experiment Results

Following conclusions can be drawn when evaluating the experiment results for finding and enforcing (m, k) constraints:

1. Finding and testing the effectiveness of (m, k) for fault injection into sensor data confirm the experiment results in [8]. The (m, k) constraint of a control task for sensor sampling data can be found empirically and can then be used to prevent the robot from failing. Thus, an (m, k) constraint exists and can be enforced to guarantee successful executions.

2. (m, k) constraints exists for the balance control task in this application. When injecting faults into motor input values in delicate tasks such as the balancing task which can only tolerate low fault rates, (m, k) can be found too, but m has to be adjusted to the fault rate. Higher fault rates require a stricter constraint.

3. (m, k) can be found and enforced not only on different control tasks, but also on different robots. In [1] a three wheeled nxt robot was used to find and test (m, k), and for this thesis a robot which balances on two wheels. This partially supports the proposition that (m, k) can be found and enforced for tasks in other systems too.

4. Increasing m increases the number of successful experiment runs, thus the higher m, the higher the quality of control.

5. Finding a certain (m, k) in successful runs, and then trying to enforce them may still lead to system fails, although the corresponding experiment set kept the robot in balance when trying to find constraints. Although (m, k) should theoretically prevent the robot from failing, it fails to do so. This means that finding and enforcing (m, k) has its limits, and it may be due to compensation overhead, or an other problem which was not found yet.

6. When multiple tasks are running, and if their empirically found (m, k) constraints are enforced concurrently, then m may need to be increased to prevent fails.

7. Only finding (m, k) doesn't suffice, it may need to be adapted to the circumstances or to different situations, e.g. if the path is thinner, a tighter constraint is needed to prevent fails.

8. D-DR is still the dominant compensation technique in most cases in terms of lowest utilization, despite different experiment settings.

To prove the first point an elliptical track was chosen, since a circular track was already used in [8]. To find candidates for (m, k), the task version constellation is recorded in a bitmap format as the experiment set for successful runs. This allows to look for the minimal

m, which is the minimal number of successful task instances in a certain window size k. Enforcing the constraint $(5, 20)$ for sensor injection with a fault rate that would cause the system to fail, kept the robot on track. This proves the effectiveness of (m, k) for arbitrary fault rates. However, the effectiveness of (m, k) was shown by recording the distances the robot achieved on the track. The same was done for all experiment runs. It turns out that the average distances do not change noticeably, they seem to change randomly. Distances should not be used to measure the quality of control in experiments with balancer, since there are multiple variables which influence the robot's distance, e.g. starting angle on the block and incline of the ground. The differences in the reached distances are marginal, although there is a trend which suggests that the tighter the constraint, the higher the distance. Using a fault rate of 80%, $(3, 20)$ reaches a distance value of 3.2, whereas $(5, 20)$ reaches a value of 10.7.

The same argument applies for injection into motor input values. In this case $(10, 16)$ was found empirically. However, as explained in the fifth point, there are still problems and open questions concerning motor injection. An (m, k) constraint was found where m is the minimal number of "1"s in a sliding window. In the runs that were done to find (m,k), the robot was stable with the amount of "1"s in the pattern. But if the (m, k) which was found in successful runs is enforced with compensation, the robot can't balance itself unless the constraint is (16,16) using a fault rate of 100%. Theoretically, any fault rate should work for $(10, 16)$, since the compensation enforces a certain amount of correct tasks, or "1"s. Only $(11, 16)$ in combination with a fault rate of 25% prevents the robot from losing balance and $(10, 16)$ is not enough to keep the robot, although it should theoretically. If the fault rate is increased, e.g. to 50%, then $(14, 16)$ needs to be chosen as the constraint, if m is lower, the experiment run will fail. If 100% fault rate is chosen, only $(16, 16)$ will prevent the robot from falling. This behavior was not observed in the sensor injection experiment. This could mean that the balancer task is very susceptible to faults and injecting faults while putting maximum strain on the compensation will lead to fails, it could also be the case that the compensation overhead is caused by following S-RE, which may slow down the execution of tasks, this could especially be crucial in delicate tasks which operate in real-time, such as the balancer. Another possibility is that the search for (m, k) candidates is flawed in some way. An answer or solution to this issue can not be given at this point. Measuring the time delay caused by the compensation and then adding this time with wait functions may allow to investigate whether the compensation overhead is the root of the problem.

In [8], the experiment for finding and enforcing (m, k) was only done for a robot which drives on three wheels, it does not have balancer task. The next step to further support the proposition that (m, k) can be found for tasks on other systems too is to find and test constraints on completely different systems, e.g. a different robot model.

The conclusion in the fourth point is supported by all experiment results. For sensor injection, increasing m always leads to an increase of the success ratio of the runs. This

is independent from the fault rate. In the same way, increasing m in motor injection also increases the amount of successful runs.

For the sixth point, faults are injected into motor and sensor data, while both tasks are protected with S-RE. $(5, 20)$ with a fault rate of 60% for the sensors and $(11, 16)$ with 25% for the motors are set as constraints and fault rates. When injecting faults only in one task, and leaving the other one without injection, then the robot never fails. However, when enforcing $(5, 20)$ and $(11, 16)$ with fault injection concurrently, the quality of control decreases, and leads to failed runs in 10% of all runs. Enforcing $(8, 20)$ and $(13, 16)$ with the same fault rates increases the quality of control, and prevents the system from failing in all cases. When there is fault injection in both tasks, then the fault of one task may influence the other task. If faults occur successively in different tasks, then the combination of the effects of the faults could lead to an unexpected fail, e.g. if both access the motors. This is the reason why one could consider choosing a tighter (m, k) constraint when multiple tasks need to be protected.

For the eighth point, we can conclude that it is not beneficial to execute D-DR all the time without further consideration, although it seems to be the dominant technique in most cases. When fault rates or interference noise are higher, the margin of benefit between D-DR and S-RE is too small. Therefore the choice between the techniques should be considered thoroughly. D-DR must use the detection version, whereas the simplest approach S-RE only needs the unreliable version. If we only consider the static compensation techniques for the simpler implementation, S-DR could also be a very good choice.

Chapter 8

Conclusion

The goal of this thesis was to verify and expand the research in [8], through first retracing and replicating the experiments which were already done, to be able to take the approaches and apply them on different experiment settings. This allowed us to reach more general conclusions about fault tolerance of control tasks and the proposed soft-error handling techniques. The methods to find and test (m, k) were originally only successfully applied for fault injection into sensor sampling data. In this thesis, the same methods were applied for fault injection into motor steering values.

In [8] (m, k) constraints were found for fault injection into sensor data, they were tested and their effectiveness was assessed through tests on a circular track. Finding the (m, k) constraint of a task allows to enforce and test it with different soft error handling techniques, which aim to minimize energy consumption, overall utilization, and execution time, which are paramount in embedded systems. The effectiveness of the presented error handling techniques is shown in [8] using the constraint $(3, 10)$, while testing different fault rates and higher m.

Finding (m, k) constraints for other experiment settings was the main objective in this thesis. To verify the previous results, $(5, 20)$ was found for an elliptical track, by injecting faults into the light sensors of a two wheeled self-balancing robot. In [8] the constraint for a circular track for the same robot was $(4, 16)$. This confirms that (m, k) constraints exist in control applications in which a limited number of errors, e.g. through noise of the environment, can be tolerated and only cause a downgrade of quality of control. It also became clear that the system may need different constraints for different situations, e.g. when there is more curvature, or when the path line is very thin.

Just to change the tracks was not enough, so fault injection into motor values was tried. First, applying the methods to find (m, k) for motor injection seemed hopeless, since the balancing task could not tolerate any errors, and fell down every time. Even very low fault rates destroyed the robot's balance. By trying to circumvent the balance control through injecting faults into the balance functions' output value, which is responsible for the strength of the signal which is sent to the motors, the task could tolerate fault

rates up to 16%. This gives us the margin to inject faults, and it was possible to find potential (m, k) candidates. $(10, 16)$ seemed to be a fitting constraint after analyzing the experiment data, and thus it was tested with different fault rates. By limiting the fault rate to 25%, which is reasonable since faults do not occur that often in reality, $(11, 16)$ was found as a stable constraint. When higher fault rates occur in the balancing task, then the constraint needs to be tighter. This behavior was not observed in the path tracing task, since its $(5, 20)$ constraint still kept the robot from going off track, despite 100% fault rate. Theoretically, the robot should not lose balance, even in the case of 100% fault rate, since the constellation of its task pattern is a pattern which would occur if $(10, 16)$ was enforced. This question still remains open and a definite answer can not be given at this point, but one could assume that it has to do with the nature of the task, since the balancing task is very susceptible to faults.

The last step for injection experiments was to inject faults into motor and sensor data concurrently. It revealed that enforcing the individual (m, k) constraints may not guarantee successful executions, the constraints possibly need to be tightened to guarantee successful executions in every run. When testing different path widths, the (m, k) constraint needed to be tightened too, e.g. the 1 cm wide line needs a $(10, 20)$ constraint to trace the path successfully, while a $(4, 20)$ constraint suffices for a 4.5 cm wide path line, and $(5, 20)$ suffices for a path width of 2 cm.

In the end, to test the effectiveness of the four proposed soft-error handling techniques, the found (m, k) constraints were tested with different fault rates and increasing m. Although it is still the dominant technique in most cases, it turns out that D-DR is not always the best choice in all cases, S-DR could be a viable trade-off between protection and overhead.

However, the model which was used in [8] and in this thesis has its downsides too. First, faults only occur in particular parts of the robot. They should ideally occur in other parts of the robot randomly too, e.g. in RAM or register, to have a more realistic model. As for experiments, the two wheeled robot is difficult to use, it requires practice to get used to setting up the robot into the same starting position every time, and sometimes a wrong starting angle distorts the experiment data. Also, the robot does not have much I/O capabilities, thus the only method to retrieve information is using the screen. Nonetheless the hardware and especially the operating system allow for high customizability and freedom, and this allows us to use the system as a tool to search, find, test and enforce (m, k) constraints.

The satellite Hitomi was touched upon in the very beginning, and the reason it crashed was a soft error in its rotation control. The NXT robot can be seen as a model for this satellite. It has a similar task to "rotation control", which is its balance control. An (m, k) constraint exists for its balancing task, and the task can tolerate faults without falling down. Finding more (m, k) constraints on different systems will hopefully give more insight to certain questions in the future, e.g. "Why can one observe certain constraints,

but not enforce them to prevent failure?". The results in this field of research, in which the aim is to minimize energy consumption while at the same time preventing failures and guaranteeing a certain quality of control, and the methods and techniques discussed in this thesis have a lot of possible areas of application in the present and in the future, e.g. in the automotive, aviation, or space craft industry. Research in this field may prevent failures in the future.

Appendix A

(m, k) Tables

(m,k)	(5,20)	(4,20)	(3,20)	(2,20)	(1,20)
60% FR	20/20	20/20	20/20	20/20	17/20
80% FR	20/20	17/20	10/20	0/20	0/20
100% FR	20/20	7/20	0/20	0/20	0/20

Table A.1: (m, k) constraints for sensor injection with fault rates of 60%, 80%, and 100%, number of successful runs out of 20

(m,k)	(16,16)	(15,16)	(14,16)	(13,16)	(12,16)	(11,16)	(10,16)
25% FR	20/20	20/20	20/20	20/20	20/20	20/20	13/20
50% FR	20/20	20/20	20/20	13/20	0/20	0/20	0/20
100% FR	20/20	15/20	9/20	5/20	6/20	2/20	4/20

Table A.2: (m, k) constraints for motor injection with fault rates of 25%, 50%, and 100%, number of successful runs out of 20

Success	Fault Rate %	correct tasks	K=16	K=17	K=18	K=19
Y	5	5027	(12,16)			
Y	10	4759	(10,16)			
Y	15	4480	(7,16)			
Y	20	4184	(5,16)			
Y	25	3941	(5,16)			
Y	30	3669	(4,16)			
Y	35	3369	(4,16)			
Y	40	3126	(4,16)			
Y	45	2934	(3,16)			
Y	50	2660	(2,14)			
Y	55	2333	(1,14)			
N	60	2114				
Y	34.9	3369	(4,16)	(4,17)	(5,18)	(5,19)
Y	36	3325	(4,16)	(4,17)	(5,18)	(5,19)
Y	37	3277	(4,16)	(4,17)	(5,18)	(5,19)
Y	38	3219	(4,16)	(5,17)	(5,18)	(5,19)
Y	39	3171	(4,16)	(4,17)	(5,18)	(5,19)
Y	40	3132	(4,16)	(4,17)	(5,18)	(5,19)
Y	41	3081	(4,16)	(4,17)	(5,18)	(5,19)
Y	42	3061	(4,16)	(4,17)	(5,18)	(5,19)
Y	43	3015	(4,16)	(4,16)	(4,17)	(5,18)
Y	44	2960	(3,16)	(4,16)	(4,17)	(5,18)
Y	45	2943	(3,16)	(3,16)	(4,17)	(4,18)
Y	46	2879	(3,15)	(3,16)	(3,16)	(3,17)
Y	47	2826	(2,15)	(3,16)	(3,16)	(3,16)
Y	48	2770	(2,15)	(3,16)	(3,16)	(3,16)
Y	49	2716	(2,14)	(3,15)	(3,15)	(3,16)
Y	50	2670	(2,14)	(3,15)	(3,15)	(3,16)
Y	51	2601	(2,14)	(2,15)	(2,15)	(3,16)
Y	52	2531	(2,14)	(2,15)	(3,15)	(3,16)
N	53	2483				

Figure A.1: Left side of the table of (m, k) candidates for the path tracing task, red means failed run, green indicates possible (m, k) candidates, and the yellow constraints were found increasing the size of k. Only contraints for $k \geqslant 16$ are relevant

K=20	K=21	K=22	K=23	K=24	K=25	K=26
(5,20)						
(5,20)						
	(5,20)					
(5,19)	(5,20)					
(5,19)	(5,20)					
(5,19)	(5,20)					
(5,19)	(5,20)					
(5,20)	(6,21)					
(5,20)	(6,21)					
(5,20)	(6,21)					
(5,20)	(6,21)					
(5,19)	(5,19)	(6,20)				
(5,19)	(5,19)	(5,20)				
(5,19)	(5,19)	(5,20)				
(4,18)	(5,19)	(5,20)				
(4,17)	(5,18)	(5,19)	(5,20)	(5,20)		
(4,17)	(4,18)	(5,19)	(5,20)	(5,20)		
(4,17)	(4,17)	(5,18)	(5,18)	(5,19)	(5,20)	
(4,17)	(4,17)	(4,18)	(5,18)	(5,19)	(5,20)	
(3,17)	(3,17)	(3,18)	(3,18)	(4,19)	(4,19)	(4,20)
(3,17)	(3,17)	(4,18)	(4,18)	(4,19)	(4,19)	(5,20)

Figure A.2: Right side of the table of (m, k) candidates for the path tracing task, red means failed run, green indicates possible (m, k) candidates, and the yellow constraints were found increasing the size of k

Success	Fault Rate %	correct tasks	K=6	K=7	K=8	K=9
Y	8.0	4138	(2,6)	(3,7)	(4,8)	(5,9)
Y	7.5	4165	(2,6)	(3,7)	(4,8)	(5,9)
Y	7.0	4185	(3,6)	(4,7)	(4,8)	(5,9)
Y	7.2	4159	(2,6)	(3,7)	(4,8)	(5,9)
Y	7.8	4147	(2,6)	(3,7)	(4,8)	(5,9)
Y	6.8	4195	(3,6)	(4,7)	(4,8)	(5,9)
Y	6.0	5005	(3,6)	(4,7)	(4,8)	(4,9)
Y	6.2	4995	(3,6)	(4,7)	(4,8)	(5,9)
Y	6.4	4989	(3,6)	(4,7)	(4,8)	(5,9)
Y	6.6	4974	(3,6)	(4,7)	(4,8)	(5,9)
Y	6.8	4970	(3,6)	(4,7)	(4,8)	(5,9)
Y	7.0	4958	(3,6)	(4,7)	(4,8)	(5,9)
Y	7.2	4949	(2,6)	(3,7)	(4,8)	(5,9)
Y	7.4	4942	(2,6)	(3,7)	(4,8)	(5,9)
Y	7.6	4934	(2,6)	(3,7)	(4,8)	(5,9)
Y	7.8	4920	(2,6)	(3,7)	(4,8)	(5,9)
Y	8.0	4910	(2,6)	(3,7)	(4,8)	(5,9)
Y	8.5	4844	(2,6)	(2,7)	(3,8)	(3,9)
Y	8.8	4870	(2,6)	(2,7)	(3,8)	(3,9)
Y	9.0	4862	(2,6)	(2,7)	(3,8)	(3,9)
Y	9.2	4849	(2,6)	(2,7)	(3,8)	(3,9)
Y	9.4	4833	(2,6)	(2,7)	(3,8)	(3,9)
Y	9.6	4826	(2,6)	(2,7)	(3,8)	(3,9)
Y	9.8	4811	(2,6)	(2,7)	(3,8)	(3,9)
Y	10.0	4802	(2,6)	(2,7)	(3,8)	(3,9)
Y	11.0	4802	(2,6)	(2,7)	(3,8)	(3,9)
Y	12.0	4704	(2,6)	(2,7)	(3,8)	(3,9)
Y	14.0	4602	(2,6)	(2,7)	(3,8)	(3,9)
Y	15.0	4547	(2,6)	(2,7)	(3,8)	(3,9)
Y	16.0	4490	(2,6)	(2,7)	(3,8)	(3,9)
N	17.0					

Figure A.3: Left side of the table of (m, k) candidates for the balancing task, red means failed run, green indicates possible (m, k) candidates

K=10	K=11	K=12	K=13	K=14	K=15	K=16
(6,10)	(6,11)	(7,12)	(8,13)	(9,14)	(10,15)	(11,16)
(6,10)	(6,11)	(7,12)	(8,13)	(9,14)	(10,15)	(11,16)
(6,10)	(7,11)	(8,12)	(9,13)	(10,14)	(10,15)	(11,16)
(6,10)	(6,11)	(7,12)	(8,13)	(9,14)	(10,15)	(10,16)
(6,10)	(6,11)	(7,12)	(8,13)	(9,14)	(10,15)	(11,16)
(6,10)	(7,11)	(8,12)	(9,13)	(10,14)	(10,15)	(11,16)
(6,10)	(7,11)	(8,12)	(9,13)	(10,14)	(10,15)	(11,16)
(6,10)	(7,11)	(8,12)	(9,13)	(10,14)	(10,15)	(11,16)
(6,10)	(7,11)	(8,12)	(9,13)	(10,14)	(10,15)	(11,16)
(6,10)	(7,11)	(8,12)	(9,13)	(10,14)	(10,15)	(11,16)
(6,10)	(7,11)	(8,12)	(9,13)	(10,14)	(10,15)	(11,16)
(6,10)	(7,11)	(8,12)	(9,13)	(10,14)	(10,15)	(11,16)
(6,10)	(6,11)	(7,12)	(8,13)	(9,14)	(10,15)	(11,16)
(6,10)	(6,11)	(7,12)	(8,13)	(9,14)	(10,15)	(11,16)
(6,10)	(6,11)	(7,12)	(8,13)	(9,14)	(10,15)	(11,16)
(6,10)	(6,11)	(7,12)	(8,13)	(9,14)	(10,15)	(11,16)
(6,10)	(6,11)	(7,12)	(8,13)	(9,14)	(10,15)	(11,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(7.13)	(7,14)	(8,15)	(9,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(7.13)	(8,14)	(9,15)	(10,16)
(4,10)	(5,11)	(6,12)	(6.13)	(7,14)	(8,15)	(8,16)
(4,10)	(5,11)	(6,12)	(6.13)	(7,14)	(8,15)	(8,16)

Figure A.4: Right side of the table of (m, k) candidates for the balancing task. Red means failed run, green indicates possible (m, k) candidates

Appendix B

Overall Utilization Data

(5,20), 10%	S-RE	S-DR	D-RE	D-DR
Sum	4747	4742	4750	4750
#u	3562	3557		
#d		1185	4750	4750
#r	1185	217	0	0
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.25	0.14	0.12	0.12

(10,20), 10%	S-RE	S-DR	D-RE	D-DR
Sum	4744	4742	4750	4750
#u	2374	2372		
#d		2370	4750	4750
#r	2370	442	0	0
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.4	0.18	0.12	0.12

(5,20), 20%	S-RE	S-DR	D-RE	D-DR
Sum	4741	4739	4744	4741
#u	3356	3555		
#d		1184	4279	4276
#r	1185	413	465	465
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.25	0.17	0.18	0.18

(10,20), 20%	S-RE	S-DR	D-RE	D-DR
Sum	4745	4745	4741	4740
#u	2375	2375		
#d		2370	4031	4740
#r	2370	838	710	237
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.4	0.23	0.21	0.16

(5,20), 30%	S-RE	S-DR	D-RE	D-DR
Sum	4745	4746	4743	4743
#u	3560	3561		
#d		1185	3363	4743
#r	1185	619	1380	613
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.25	0.2	0.29	0.21

(10,20), 30%	S-RE	S-DR	D-RE	D-DR
Sum	4745	4743	4747	4747
#u	2375	2373		
#d		2370	3257	4747
#r	2370	1227	1490	753
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.4	0.29	0.3	0.23

(15,20), 10%	S-RE	S-DR	D-RE	D-DR
Sum	4745	4742	4748	4743
#u	1190	1187		
#d		3555	3128	4743
#r	3555	670	1620	271
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.55	0.21	0.32	0.16

(11,16), 10%	S-RE	S-DR	D-RE	D-DR
Sum	4745	4742	4744	4744
#u	1485	1485		
#d		3257	4744	4744
#r	3260	602	961	185
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.51	0.2	0.26	0.14

(15,20), 20%	S-RE	S-DR	D-RE	D-DR
Sum	4748	4740	4746	4747
#u	1190	1185		
#d		3555	2256	4747
#r	3558	1311	2490	890
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.55	0.31	0.42	0.25

(11,16), 20%	S-RE	S-DR	D-RE	D-DR
Sum	4745	4743	4745	4745
#u	1485	1485		
#d		3258	2622	4745
#r	3260	1219	2123	762
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.51	0.3	0.38	0.23

(15,20), 30%	S-RE	S-DR	D-RE	D-DR
Sum	4745	4742	4743	4746
#u	1190	1187		
#d		3555	1607	4746
#r	3555	1824	3136	1614
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.55	0.4	0.5	0.35

(11,16), 30%	S-RE	S-DR	D-RE	D-DR
Sum	4744	4742	4745	4748
#u	1485	1485		
#d		3257	1852	4748
#r	3259	1665	2893	781
unrel	100	100		
det		120	120	120
rel	700	700	700	700
	0.51	0.36	0.48	0.24

Figure B.1: Overall utilization data for all constraints and fault rates.

List of Figures

List of Tables

List of Algorithms

Bibliography

[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] Japan Aerospace Exploration Agency (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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Kuan-Hsun Chen, Björn Bönninghoff, Jian-Jia Chen, and Peter Marwedel. Compensate or ignore? Meeting Control Robustness Requirements through Adaptive Soft-Error Handling. In *Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, Santa Barbara, CA, U.S.A., June 2016. ACM.

[9] Kuan-Hsun Chen. LCTS slides for the presentation of the paper "Compensate or Ignore? Meeting Control Robustness Requirements through Adaptive Soft-Error Handling".

[10] 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.

[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] 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.

[13] 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.

[14] Hands-On Session 2: Fault-Injection based Assessment of Software-Implemented Hardware Fault Tolerance, Winter School on Operating Systems, February 22-26, 2016, Graz/Austria, 2016.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[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] 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.

[22] NXTway-GS Building Instructions. `http://lejos-osek.sourceforge.net/NXTway-GS_Building_Instructions.pdf`.

[23] Dennis Nahberger. Fehlerbehandlung in echtzeitfähigen Steueranwendungen. Master's thesis, TU Dortmund, 2012. Diplomarbeit.

[24] Y.Yamamoto. Two wheeled self-balancing R/C robot controlled with a Hitechnic Gyro Sensor. `http://lejos-osek.sourceforge.net/nxtway_gs.htm`, 2010.

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

[26] NXTway-GS (Self-Balancing Two-Wheeled Robot) Controller Design. `http://lejos-osek.sourceforge.net/NXTway-GS_Building_Instructions.pdf`.

[27] 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.

[28] J. R. Sklaroff. Redundancy management technique for space shuttle computers. *IBM Journal of Research and Development*, 20(1):20–28, Jan 1976.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kentlich gemacht habe.

Dortmund, 23. März 2017

Mikail Yayla