European Space Agency

ESA Summer of Code in Space 2017
Final Report

Software-based Fault Tolerance:
Implementation of a Fault Tolerant Rate Monotonic Scheduler

Mikail Yayla
September 30, 2017

Mentors:
Dr. Gedare Bloom
M.Sc. Kuan-Hsun Chen
Dr. Joel Sherill

Contents

Chapter 1

Introduction

1.1 Motivation

Mobile and embedded systems are susceptible to transient faults in the underlying hardware [1], which may occur due to rising integration density, low voltage operation and environmental influences such as radiation and electromagnetic interference. Transient faults may alter the execution state or incur soft-errors. Bitflips are a direct cause of transient faults, and occur in register, processor, main memory, and 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 in [2] a bitflip occured in the satellite's rotation control, which made it rotate uncontrollably; it broke into several pieces. The financial damage was severe; a few hundred million Euros. To prevent such catastrophies, one way is to utilize software-based approaches such as redundant execution and error-correction code [3, 4, 5, 6, 7]. Yet, trivially adopting redundant execution or error correction code may lead to significant computational overhead, e.g., when N+1 redundancy is used. Due to spatial limitation and timeliness, skillfully adopting software-based fault-tolerance approaches to prevent system failure due to transient faults is not a trivial problem.

Instead of over-provisioning with extra circuit or execution time, sometimes errors can be tolerated; there may be no need to protect the whole system. Due to the potential inherent safety margins and noise interference, control applications might tolerate a limited number of errors with a downgrade of control performance without leading to an unrecoverable system state. In previous studies, techniques have been proposed for delayed [10, 11] or dropped [12, 13] signal samples. Chen et al. [8] explored the fault tolerance of a LegoNXT path-tracing control application. To prevent the system from mission failures, they proposed to use the (m, k)-firm real-time task model to quantify the robustness of control tasks, which means that m out of k consecutive task instances need to be correct,

called (m, k) robustness requirement, using which they provided compensation techniques to manage the expensive error correction to avoid overprovision.

1.2 Project Goal

The purpose of this project is to provide an implementation of the techniques in [8] on the real-time operating system RTEMS [ADD]. In the previous study, the application solely runs on a LegoNXT robot using nxtOSEK, and is not portable. An implementation in RTEMS allows us to port the application to several other platforms; we can test the techniques on other hardware, e.g., Raspberry Pi, to reach more general conclusions about fault tolerance of control tasks and the proposed soft-error handling techniques. The application including the software techniques may be an option for space vehicles to manage redundant executions in the future.

1.3 Report Structure

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

The system model and notations, which is necessary to describe the issues in [8], are introduced in the next section. Task versions, the robustness requirement (m, k), and patterns used to decide when to execute which task version are also covered. The soft-error handling techniques S-RE, S-DR, D-RE, and D-DR will be the central theme in section three.

Chapter 2

Background

In this chapter, transient faults and their impact on systems will be introduced. An explanation of the application model, task versions, execution patterns, and the (m, k) requirement follows.

2.1 Causes of Transient Faults

Transient faults can occur in hardware due to rising integration density, low voltage operation and environmental influences such as radiation and electromagnetic interference [1]. By lowering the voltage operation and integration density, the probability of faults increases, because less electrons are used to distinguish the state of the hardware. 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], causing bitflips, which means that a zero turns into a one or the other way around. The consequences of bitflips are difficult to predict, in the worst case they could lead to irrecoverable system failure; they are dangerous and their effects need to be kept at bay.

2.2 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, in case the deciding "1" is not affected.

In the worst case, faults can cause system failure and lead to a system crash. A flipped bit in variable that points to a certain location may result in accessing wrong code, or invalid memory locations, potentially leading to a system crash.

On the other hand, some faults may remain unnoticed. If a variable in the memory is affected and its value is used for calculations, computation could proceed as planned but with faulty output values.

2.3 Fault Model in the Fault Tolerant Scheduler

The scheduler is designed to handle faults which appear in certain parts of the task entity, i.e., variables in main memory, register, cache, and during data transfer of these values between hardware components. The system is protected from the effects of these faults, i.e., incorrect calculations, for example when there is lack of or faulty control task output; it can be detected and corrected. The main application of the FTS are control tasks such as path tracking, steering, stabilization control, etc., since we can guarantee correct output values. The FTS can only protect from faults which do not cause an unrecoverable system state. For example, if a fault occurs in the instruction code of the scheduler itself, the system may crash. The FTS cannot prevent the system from the effects of such kind of faults. However, from the execution versions which are provided by the RTEMS user, we expect that detection and correction are always (or with very high probability) possible; all errors should be detected and corrected successfully. In Silent Data corruption for example, when errors remain unnoticed but computation continues as usual, it would be problematic for the system if the detection or correction rate would not be sufficiently high; it could lead to mission failure, such as in the case of satellite Hitomi.

Chapter 3

System Model

In this section models and notations used in this report are explained.

3.1 Control Application Model

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

A task is available in three versions with different execution times: unreliable version τ_i^u with execution time c_i^u, error detection version τ_i^d with c_i^d, and error correcting version τ_i^c with c_i^c. The least protected one is τ_i^u, it allows incorrect output, and only protects from errors that lead to system crash by affecting the remaining system.

In the literature, several fault tolerant software techniques require additional execution time for error handling, e.g., special encoding of data [16], control flow checking [15], and majority voting [17]; we assume $c_i^u < c_i^d < c_i^c$ holds.

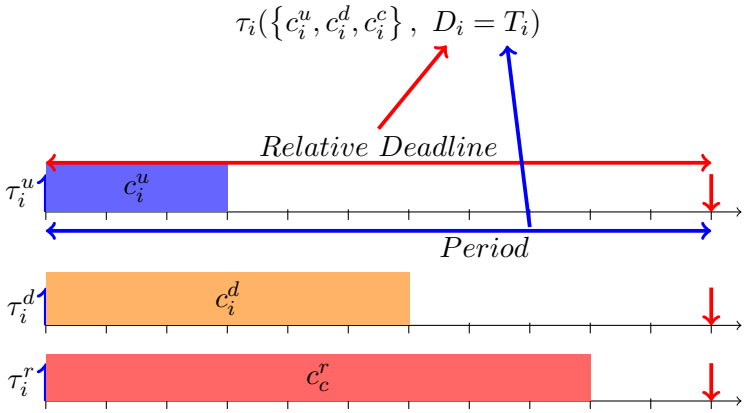

Figure 3.1: The different task versions and their execution times relative to each other [9]

3.2 (m, k) robustness requirement

To quantify the inherent tolerance of tasks to recover from previous instance's lack of or faulty output, we use the (m_i, k_i) robustness requirement. m out of any k consecutive task instances have to be correct for the control application to function without mission failure. m_i and k_i are both positive integers and $0 < m_i \leq k_i$.

For example, consider a control task controlling specific steering actions of a vehicle. Its robustness requirement could be that there are at least two correct in *every* three instances, denoted as $(2, 3)$. Fig. 3.2 contains the information whether faults were injected into instances of τ_i. If a fault was injected, we add "0" to the bitmap, otherwise we add "1", this is done for all instances of τ_i. In the experiments, the iterative way of checking is called *sliding window*, which is defined as follows: we first check the first three bits $\{1, 1, 0\}$, then the second three $\{1, 0, 1\}$, the third three $\{0, 1, 1\}$, etc. All such sets in the bitmap have at least two correct task instances, we say the $(2, 3)$ requirement is fulfilled. If the $(2, 3)$ requirement is not fulfilled, e.g., $\{0, 0, 1\}$ occurs in the bitmap, the vehicle may steer in a wrong direction and possibly cause an accident.

1	1	0	1	1	0	1	1

Figure 3.2: The bitmap of a task τ_i for eight executed instances. "1" represents instances without a fault, and "0" for instances with a fault.

3.3 Schedulability and Scheduling

The FTS manages the executions of the different task versions by adopting RM scheduling. τ_1 has the highest priority and τ_n the lowest. If all tasks meet their deadlines while (m_i, k_i) holds for each τ_i, then the schedule is feasible. For all the applied scheduling strategies, their schedulability tests can be found in [8].

Chapter 4

Soft-Error Handling Techniques

The allocation of the reliable executions for a task τ_i to enforce (m_i, k_i) requirements using (m, k) patterns is discussed in this section. Pattern-Based Reliable execution is introduced first; it forms the basic concepts on which the dynamic compensation techniques build.

4.1 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 (S-RE), if only m out of k instances need to be correct, then only m of those tasks need to be executed using correction. To 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 $\Phi_i = \{\phi_{i,0}, \phi_{i,1}, \ldots \phi_{i,(k_i-1)}\}$ which satisfies the following properties: $\phi_{i,j}$ is a reliable instance if $\phi_{i,j} = 1$ and a unreliable instance if $\phi_{i,j} = 0$, while $\sum_{j=0}^{k_i-1} \phi_{i,j} = m_i$ [8].

A robustness requirement could be given as $(3, 5)$, then one way to define the (m, k)-pattern is $\Phi = \{0, 1, 0, 1, 1\}$. Executing all instances denoted as "1" with τ_i^c and using τ_i^u to execute the instances marked with a "0" satisfies the $(3, 5)$ requirement; directly executing the reliable versions this way is called S-RE.

Static Detection and Recovery (S-DR) gives a try with a fault-detecting version when there is a "1" in the pattern, and immediately executes a correction version if an error is detected. The following example highlight the difference between the two techniques: For S-RE and S-DR, the (m, k)-pattern $\Phi = \{0, 1, 0, 1, 1\}$ with the corresponding (m, k) requirement $(3, 5)$ is specified for a task τ_i. S-RE directly executes τ_i^c whenever there is a "1" in the pattern, and executes τ_i^u if there is a 0. $\{\tau^u, \tau^c, \tau^u, \tau^c, \tau^c\}$ is the schedule for this specific task; the sum of the execution times for one completion of the string is $2c^u + 3c^c$.

On the other hand, S-RE first gives a try with the detection version of a task whenever the index of the pattern points to "1". If an error is detected when executing τ^d, the

correction version of the task is executed immediately. A possible schedule, if we assume an error occurs on the second instance, would be: $\{\tau^u, \tau^d + \tau^c, \tau^u, \tau^d, \tau^d\}$. If errors occur in every instance, in the worst case, the sum of the execution times is $2c^u + 3(c^d + c^r)$. Figure 4.1 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 correction version immediately after the detection version.

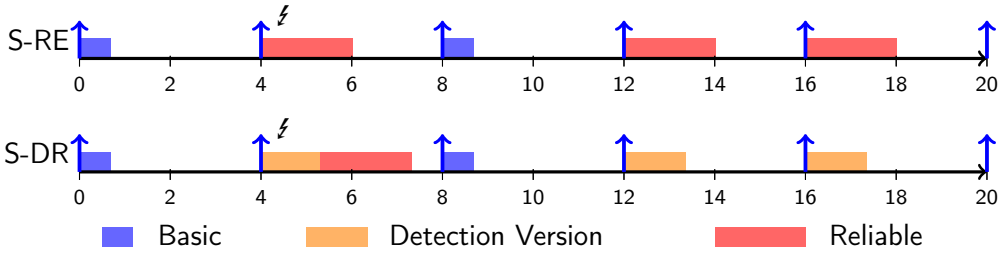

Figure 4.1: Examples for the static approaches for $(3, 5)$ with the pattern $\{0, 1, 0, 1, 1\}$, adapted from [9]

4.2 Dynamic Compensation

This part of the report first discusses the main principle behind dynamic compensation, which is postponing. The algorithm is presented which illustrates dynamic compensation in pseudo-code. At the end of this chapter examples outline 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 implemented in RTEMS.

Soft-errors happen randomly from time to time, and the likelihood of their occurrence is generally not very high. Minimizing the executions of τ^c tasks is one of the main goals in these techniques, as τ^c is very costly considering execution time and energy consumption. We only want to execute τ^c if it is absolutely necessary.

Dynamic Compensation enhances S-RE and S-DR by making decisions on-the-fly. The main idea is to postpone the moment the system enforces the (m_i, k_i) requirement by exploiting the correctness of successful executions of task instances. In the worst case, if all unreliable task versions are wrong, the system will follow the execution pattern. However, in dynamic compensation, all instances that are marked with "0" in the (m, k)-pattern need to be executed with the error τ^d to detect errors.

4.2.1 Postponing the Pattern

We suppose that a static pattern Φ_i is given. Φ_i is a binary string, containing the information about which version of a task should be executed. It was stated 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 tau^d instances, and then exploits their correctness by postponing the adoption of the (m, k)-pattern. 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. We can think of it as inserting S 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 pattern $\{0, 1, 0, 1, 1\}$ 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 and no faults occur. Because of the successful execution of two task instances, denoted as S in the pattern, the string at $t = 2$ now lacks the last two "1"s from the original pattern. These two are pushed out of the string, and thus the adoption of the original pattern is 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 means that only one more task instance needs to be correct for the (m_i, k_i) requirement to be fulfilled. S instances are always executed with τ_i^d to detect errors. S can be explained as "gave a try with detection version; got away with it (without an error occuring)".

4.2.2 Tolerance Counters

Because we can only give a try with a detection version if there is a "0" in the pattern, we need to partition it. Replenishment counters 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 rule is to divide the original 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. 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 $a_{i2} = 2$.

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

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.

4.2.3 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.

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 report.

4.3 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 4.2, in which D-DR performs the lowest regarding overall utilization among them.

In Figure 4.2 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.

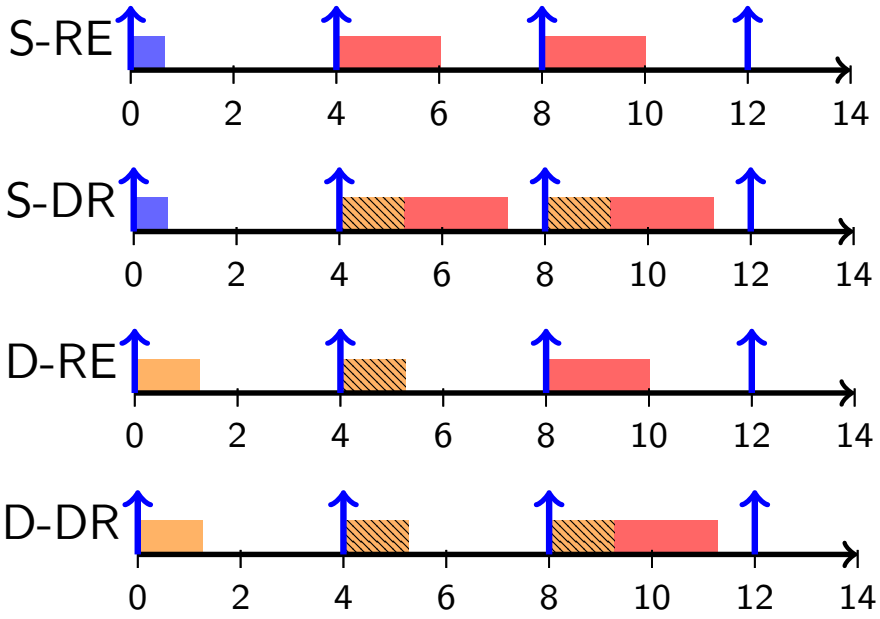

Figure 4.2: This example illustrates the different techniques. (m_i, k_i) is $(2, 3)$ and the static pattern is $\Phi = (0, 1, 1)$. Soft-errors happen in the second and third instance and are marked with stripes. The Blue block is unreliable, the orange block is the version with detection, and red block is reliable. Adapted from [8].

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.