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: Gedare Bloom, Kuan-Hsun Chen, 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 hardware or execution time, sometimes errors can be tolerated; there may be no need to protect the whole system. Due to the potential inherent safety margins and noise interference, control applications might tolerate a limited number of errors with a downgrade of control performance without leading to an unrecoverable system state. In previous studies, techniques have been proposed for delayed [8, 9] or dropped [10, 11] signal samples. Chen et al. [12] 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 [12] 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 fault tolerance 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 the first chapter. The system model and notation is introduced in chapter two. Task versions, the robustness requirement (m, k), and patterns used to decide when to execute which task version are also covered. In chapter three, the soft-error handling techniques S-RE, S-DR, D-RE, and D-DR will be the central theme.

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. In the worst case, bitflips 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 may have no noticeable consequences, because the flipped bit may 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 a 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 Faults 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, such as sensor sampling data. The system is protected by the FTS scheduler from incorrect calculations incurred by faults, 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, the FTS can not protect the system if a fault occurs in the instruction code of the scheduler itself, the system may crash.

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. If silent data corruption occurs - when errors remain unnoticed but computation continues as usual - it would be fatal 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 and Notation

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 [13], control flow checking [14], and majority voting [15]; we assume $c_i^u < c_i^d < c_i^c$ holds.

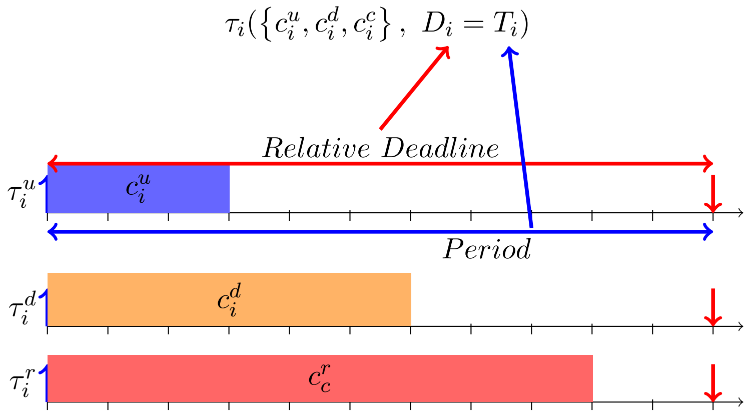

Figure 3.1: Possible setting of the task versions and their execution times, adapted from [16]

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.

$$\boxed{1 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1}$$

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 [12].

Chapter 4

Implemented Compensation Techniques

The techniques which decide when to execute which task version for τ_i to enforce (m_i, k_i) requirements using (m, k) patterns is discussed in this section.

4.1 Static Pattern Based Execution

If only m out of k instances need to be correct, then, to guarantee the correctness, only m of those tasks need to be executed using τ_i^c. To have an "execution plan" of when to execute which task version, the (m, k)-pattern [17, 18] is introduced.

Definition 1: The (m, k)-pattern of task τ_i is a binary string $\Phi_i = \left\{\phi_{i,0}, \phi_{i,1}, \dots \phi_{i,(k_i-1)}\right\}$ which satisfies the following properties: $\phi_{i,j}$ is a correction instance if $\phi_{i,j} = 1$ and a unreliable instance if $\phi_{i,j} = 0$, while $\sum_{j=0}^{k_i-1} \phi_{i,j} = m_i$ [12].

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 executing the instances marked with a "0" using τ_i^u satisfies the $(3, 5)$ requirement. Directly executing τ_i^c this way is called Static Reliable Execution (S-RE). $\{\tau^u, \tau^c, \tau^u, \tau^c, \tau^c\}$ is the schedule for this specific task; the sum of the execution times for one completion of the string is $2c^u + 3c^c$.

An improvement of S-RE, called 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 τ_i^c if an error is detected. If we assume an error occurs on the second instance, a possible schedule is $\left\{\tau^u, \tau^d + \tau^c, \tau^u, \tau^d, \tau^d\right\}$ for the pattern above. If errors occur in every instance, then in the worst case, the sum of the execution times is $2c^u + 3(c^d + c^c)$.

Figure 4.1 shows S-RE 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 τ_i^c, or the error will be tolerated, because the (m, k) requirement is always fulfilled.

In the second diagram with S-DR, the system follows the pattern as well, but always gives a try with τ_i^d first if the bit in the pattern is a "1". An error occurs in the second instance, so the system has to execute τ_i^c immediately after the detection version.

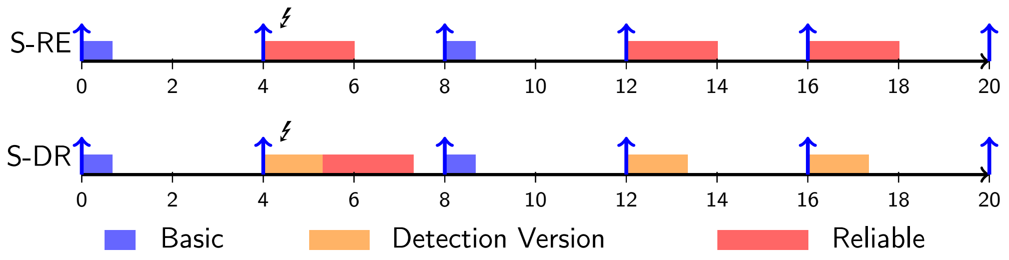

Figure 4.1: Examples for the static approaches for $(3, 5)$ with the pattern $\{0, 1, 0, 1, 1\}$, adapted from [16]

4.2 Dynamic Compensation

This part of the report first discusses the main principle behind dynamic compensation: postponing the moment the system adopts the (m, k) pattern. Then 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 dynamic approaches. This chapter concludes the basics which are necessary to understand the techniques and the underlying principles which were implemented in RTEMS.

Soft-errors happen randomly from time to time, and the likelihood of their occurrence is not very high. Minimizing the executions of τ^c tasks is one of the main goals in these techniques, as τ^c is 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 τ_i^d executions. In the worst case, if all τ_i^d are wrong, the system will adopt the execution pattern. To detect errors in dynamic compensation, all instances that are marked with "0" in the (m, k)-pattern need to be executed with τ^d.

4.2.1 Postponing the (m,k)-Pattern

We suppose that a static pattern Φ_i is given, which is a binary string, containing the information about which version of a task to execute. 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 (i.e., correct) executions of τ^d instances, and then exploits their correctness by postponing the adoption of the (m, k)-pattern by the number of successful executions. 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 "0", as only "0" gives 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 put 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 τ_i^d 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 the two partitions $\{0, 0, 1\}$ and $\{0, 1, 1\}$. The rule is to divide the original pattern into smaller patterns that start with 0 and end with 1.

After the partitioning, the number of partitions are counted and their number is saved in p_i. The index of the task is denoted as i, and j describes the current partition in the pattern. Counter o_{ij} describes the number of τ_i^u in each partition and a_{ij} counts the number of τ_i^c instances in each partition. 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 $p_i = 2$, $o_{i1} = 2$, $a_{i1} = 1$, $o_{i2} = 1$, $a_{i2} = 2$; we define $O_i = \{2, 1\}$ and $A_i = \{1, 2\}$. There are two τ_i^u instances in the first partition, $o_{i1} = 2$ and one τ_i^c $a_{i1} = 1$, whereas in the second partition, there is one unreliable $o_{i2} = 1$ instance and there are two τ_i^c instances $a_{i2} = 2$.

If o_{ij} is used up by errors, the system then has to follow the original pattern. To model that kind of switching behavior, a mode indicator Π is used, 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 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 be set to safe. This causes the system to just execute the task instances with τ_i^c. 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 τ_i^d.

4.2.3 Algorithm for Dynamic Compensation

The algorithm for dynamic compensation works on one of the partitions of an (m, k)-pattern. At the beginning the index j is passed while the mode is set depending on whether the tolerance counter o_{ij} is depleted. If Π is in tolerant mode, the algorithm executes τ_i^d which checks whether a fault was detected. If a fault is detected, then the tolerance counter o_{ij} is decreased by one.

If o_{ij} is fully depleted, equal to zero, then the system has is set to safe mode. a_{ij} stores the number of correction version 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 $l = 0$, the task is set back to tolerant mode and the index j is incremented; the next partition is processed by the algorithm. When the system is in safe mode, there are two different strategies available which can be pursued. Although the system is in safe mode (o_{ij} is already depleted), Dynamic Detection and Recovery (D-DR) will always execute τ_i^d of the task first. Thus the system still gives a try with τ_i^d and only executes τ_i^c if an error occurs in τ_i^d.

Dynamic Reliable Execution (D-RE) on the other hand will always execute τ_i^c directly if the system is in safe mode.

```
1: **procedure** dyn_Compensation($mode\ \Pi, index\ j$)
2: **if** $\Pi$ is $tolerant\_mode$ **then**
3:     $result$ = execute($\tau_i^d$);
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 $\Pi$ to $safe\_mode$;
9:             Set $\ell$ to $a_{i,j}$;
10:        **end if**
11:    **end if**
12: **else**
13:    either Detection_Recovery() or Reliable_Execution();
14:    $\ell = \ell - 1$;
15:    **if** $\ell$ is equal to 0 **then**
16:        Set $\Pi$ to $tolerant\_mode$;
17:        $j = (j + 1) \bmod k_i$;
18:    **end if**
19: **end if**
20: Update_Age($\mathbb{O}_i$);
21: **end procedure**
```

Algorithms 4.1: Dynamic compensation of task τ_i with (m_i, k_i), adapted from [12]

4.3 Summary of Soft-Error Handling techniques

In this work four soft-error handling techniques are implemented. They are ordered by their overall utilization, beginning with the highest value. D-DR shows the lowest overall utilization out of all proposed techniques [12]. To give a quick overview, the techniques are presented one below the other.

In Figure 4.2, in the first diagram, the system uses S-RE and just follows the pattern $\{0, 1, 1\}$ for the $(2, 3)$ requirement.

When using S-DR, the system executes τ_i^u in the first instance, since there is a "0" in the pattern. In the second and third task instance, it executes τ_i^d for every "1" in the pattern and immediately releases τ_i^c when an error is detected.

When D-RE is used, the system gives a try with τ_i^d in the first task instance. The task turns out to be correct, so the system still has one chance to be wrong, thus, in the second task instance, it gives a try with τ_i^d again. 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 τ_i^c 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 τ_i^d, although the $(2, 3)$ requirement would be broken if an error occurred. The system detects an error and executes τ_i^c afterwards in the third instance.

- Pattern-Based Execution
  - S-RE: Runs task version τ_i^c for instances marked as "1", τ_i^u otherwise.
  - S-DR: Runs task version τ_i^d for instances marked as "1" and recovers executing τ_i^c if an error is detected in τ_i^d.
- Dynamic Compensation
  - D-RE: Runs execution version τ_i^c if the tolerance counter is depleted, τ_i^d otherwise.
  - D-DR: Runs execution version τ_i^d if the tolerance counter is depleted; runs execution version τ_i^c for recovery if an error is detected in τ_i^d in this case.

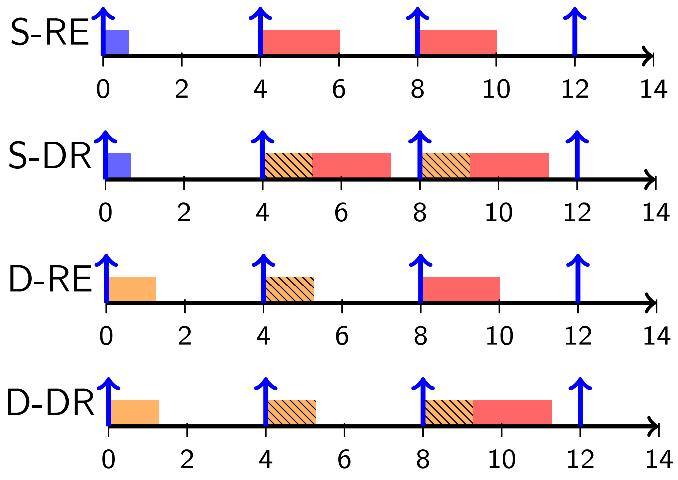

Figure 4.2: (m_i, k_i) is $(2, 3)$ and the static pattern is $\Phi = (0, 1, 1)$. Soft-errors occur in the second and third instance (marked with stripes). Blue block: τ_i^u, orange block: τ_i^d, red block: τ_i^c. Adapted from [12].

Chapter 5

The Fault Tolerant Scheduler integrated in RTEMS

In this chapter, the inner workings of the FTS are presented. Before reading this chapter, it is recommended to read the chapter about the rate-monotonic scheduler in [rm-TUT] first.

5.1 FTS Scheduler Workflow

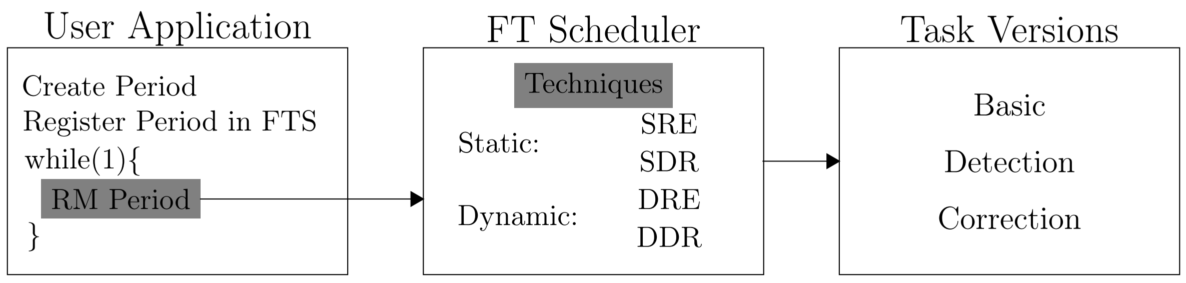

Figure 5.1: The flow diagram of the FTS. The RM Period calls the FT Scheduler

Before using the FTS, the user has to implement the three task versions, which are basic version, detection version, and correction version. When calling the function which creates the period object, the period's ID is used to register the period for protection in the FTS. After creation, the period has to be implemented as specified in [rm-TUT], using a while-loop and calling the period function in every iteration of the while-loop.

Every time the period function is called, the rate monotonic scheduler calls the FTS. The FTS then decides which version of the task to create and release. The decision is made based based on the configuration of the FTS, e.g., fault-tolerance technique, (m, k) requirement, and using R- or E-pattern.

All task versions that were started within one period are deleted at the beginning of the next period, preventing overload situations.

5.2 Implementation of the Fault-tolerance Techniques

This section is about the implementation of the fault-tolerance techniques in RTEMS. The static approaches SRE and SDR use the predefined (m, k)-pattern to decide which version of the task to execute, whereas the dynamic approaches DRE and DDR calculate the tolerance counters based on the (m, k)-pattern.

5.2.1 R- and E-patterns

There are two types of patterns available for the user, i.e., R-patterns and E-patterns [ADD1]. An R-pattern contains all "0"s at the beginning and all "1"s at the end, such as in $\{0, 0, 0, 1, 1, 1, 1\}$. In E-patterns, the "1"s and "0"s are distributed evenly. The pattern's correctness proofs of complying to an (m, k) requirement can be found in [ADD2] and [ADD3]. The pattern is read by the FTS from left to right and top to bottom, if we assume that bytes are on top of each other, such as in the figure below[DRAW FIG]. When using SRE or SDR, if the FTS is at the last byte and bit of the pattern, it starts reading the first bit in the pattern again.

5.2.2 Pattern Iteration in SRE and SDR

In the FTS, SRE and SDR follow the given static (m, k)-pattern to execute task versions while satisfying (m, k) requirements. To be able to adopt the given (m, k)-pattern, the three parameters $uint8\_t*\ \ pattern\_start$, $uint8\_t*\ \ pattern\_end$, and $uint8\_t\ \ max\_bitpos$ can be specified by the user, or are generated by the FTS automatically. The starting and ending address of the pattern is specified by $pattern\_start$ and $pattern\_end$, which are pointers to a $uint8\_t$ variable. The current byte-position is stored in $curr\_pos$, whereas the single bit position in a certain byte is stored in bit_pos. If the pattern should end in midst of a byte, the variable max_bitpos can be specified (0 for the first right bit, and 7 for the most left bit); if it is not specified, max_bitpos will be initialized as 7, which means that the first position (from the left) is the last bit to be read.

The FTS iterates through this pattern with bitwise operations. The current byte's current bit is retrieved by bitwise AND. For example, if $\{10111011\}$ is the current byte, and bit_pos is 3, the red "1" tells us to execute a reliable version next. We do the operation 10111011 & 00010000 to decide which task version to execute.

If current iteration step is in the last byte's last bit, the current byte will be set to the beginning address of the pattern, and bit_pos is set to zero.

5.2.3 DRE and DDR

In DRE and DDR, the FTS does not follow the (m, k)-pattern, but calculates the tolerance counters based on the (m, k)-pattern, and then executes the algorithm [4.2.3]. The FTS first checks the pattern type of the (m, k)-pattern. In case of R-pattern, there are only two partitions, and thus only two tolerance counters o_i and a_i. For E-patterns, the FTS divides the (m, k)-pattern into sub-patterns which begin with "0" and end with "1". At the beginning of each period, the FTS checks whether the tolerance counters are depleted. If they are depleted, they get replenished again.

5.3 Fault Injection and Detection to test the FTS in RTEMS

5.3.1 Fault Injection Mechanism in RTEMS

To simulate the occurrence of faults, a fault injection mechanism is needed. The fault rate is specified as $p\%$ per task. Then, an integer seed variable is chosen, and the random number generator in C needs to be initialized with it by calling $srand(seed)$ once. This way the $rand()$ function gives us a sequence of random numbers when it is called sequentially. The sequence is the same for every seed value, i.e. $seed = 5$ gives us a certain sequence of random numbers, and $seed = 6$ a different sequence, but the sequences are reproducible. For example, the fault rate could be specified as 3%. Then a predefined number of random values in $[0, 100]$ is created and the sequence is stored, as described in the function $rand\_nr\_list()$ below.

To decide whether a fault should be injected, we retrieve one value out of the array. Only if this random value is smaller than the fault rate, a fault is injected. The value random value is retrieved by the function $get\_rand()$ below, and the value $rand\_count$ specifies which random value has already been used. The function $fault\_statusget\_fault(...)$ checks whether the random number is smaller than the fault rate with $rand\_nr \leq fault\_rate$ and returns the fault status. The decision to inject a fault is determined by calling $get\_fault(get\_rand())$, which returns the fault status. When testing the application with a control task in the future, a certain control value can be manipulated every time $get\_fault()$ returns a fault, e.g. the control value could be set to the maximum possible value to simulate the occurrence of a fault.

```c
void rand_nr_list(void)
{
  for ( uint16_t i = 0; i < NR_RANDS; i++ )
  {
    rands_0_100[i] = rand() / (RAND_MAX / (100 + 1) + 1);
  }
  return;
}

uint8_t get_rand(void)
{
  return rands_0_100[rand_count++];
}

fault_status get_fault(uint8_t rand_nr)
{
  if ( fault_rate == 0 )
  {
    return NO_FAULT;
  }
  else if ( rand_nr <= fault_rate )
  {
    faults++;
    return FAULT;
  }
  else
  {
    return NO_FAULT;
  }
}
```

5.3.2 Implementation of Fault Detection in RTEMS

We now know how the occurrence of faults is simulated in the Fault Tolerant Scheduler. To test the fault-tolerance techniques, an approach to detect errors is needed.

As explained [inSec], the user implements the three execution versions of a task. This means that the users implement their own fault detection and correction. At the end of each detection version, the task detects an error (e.g. by executing the calculation in the task twice, and then comparing the results). The fault status is then sent to the FTS. The detection version has to call the function $fault\_detection\_routine(rtems\_id\ \ id, fault\_status\ \ fs)$ which processes the error handling. For example, in case a fault occurred when executing a detection version with SDR, this function will trigger the release of a reliable task version immediately. The detection routine handles faults in the dynamic techniques in a similar way.

5.4 Example: How to use the FTS Scheduler

The three execution versions have to be implemented first. The task argument has to be initialized as $rtems\_task\_argument\ \ argument$, because the FTS will use this argument to pass the period ID to the task versions. The detection version has to call the detection routine $fault\_detection\_routine(id,\ \ fs\_T1)$ at the end of its activity. For example, the three versions could print the ID of their period as follows:

```
/* Basic task version */
rtems_task BASIC_V(
  rtems_task_argument argument
)
{
  rtems_id id = *((rtems_id*)argument);
  printf("Basic: ID of my Period is %i\n", id);
  rtems_task_delete( rtems_task_self() );
}

```

```c
/* Detection task version */
rtems_task DETECTION_V(
  rtems_task_argument argument
)
{
  rtems_id id = *((rtems_id*)argument);
  printf("Detection: ID of my Period is %i\n", id);

  /* Check whether a fault occured */
  fault status fs_T1 = get_fault(get_rand());

  /* Call fault detection routine at the end of the detection version activity */
  fault_detection_routine(id, fs_T1);
  rtems_task_delete( rtems_task_self() );
}

/* Correction task version */
rtems_task CORRECTION_V(
  rtems_task_argument argument
)
{
  rtems_id id = *((rtems_id*)argument);
  printf("Correction: ID of my Period is %i\n", id);
  rtems_task_delete( rtems_task_self() );
}
```

We then take a look at the configuring of the FTS, all the information needs to be defined before the period is registered for in the FTS.

```c
/* last byte of (m,k)-pattern */
uint8_t end_p = 0;
/* first byte of (m,k)-pattern */
uint8_t begin_p = 0;
/* store the pointers */
uint8_t *p_s = &begin_p;
uint8_t *p_e  = &end_p;
/* Define maxbit */
uint8_t maxbit = 7;
/* RTEMS task pointers */
rtems_task *basic;
rtems_task *detection;
rtems_task *correction;
/* set (m,k) */
uint8_t m = 12;
uint8_t k = 16;
/* Set fault tolerance technique and (m,k)-pattern type */
fts_tech curr_tech = DDR;
pattern_type patt = R_PATTERN;
```

Then, we look at how a period can be registered for protection in the FTS. We first use the Init task to create and start the task which runs the while-loop, which in turn calls the period function. A possible implementation of the task containing the while-loop is shown below.

```
/* Task 1 contains the while-loop */
rtems_task FTS_TEST(
  rtems_task_argument unused
)
{
  rtems_status_code                          status;
  rtems_id                                   RM_period;
  rtems_rate_monotonic_period_status         period_status;

  /* Task entry points */
  basic = &BASIC_V;
  detection = &DETECTION_V;
  correction = &CORRECTION_V;

  /* Create a period and register the task set for RM-FTS */
  status = rtems_rate_monotonic_create_fts( Task_name, &RM_period,
   m, k, curr_tech, patt, p_s, p_e, maxbit, basic, detection, correction);

  while (1)
  {
    status = rtems_rate_monotonic_period( RM_period, 100 );

    if( status == RTEMS_TIMEOUT )
    {
      printf("\nPERIOD TIMEOUT\n");
      break;
    }
  }

  status = rtems_rate_monotonic_delete( RM_period );
  if ( status != RTEMS_SUCCESSFUL )
  {
      printf( "rtems_rate_monotonic_delete failed with status of %d.\n", status );
      exit( 1 );
  }
  status = rtems_task_delete( selfid );    /* should not return */
  printf( "rtems_task_delete returned with status of %d.\n", status );
  exit( 1 );
};
```

List of Figures

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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.