European Space Agency

ESA Summer of Code in Space 2017
Final Report

Software-based Fault Tolerance:
Adding Fault Tolerance Feature in the Rate Monotonic Scheduler

Mikail Yayla
September 30, 2017

Mentors: Gedare Bloom, Kuan-Hsun Chen, Joel Sherill

Contents

1 Introduction **2**

1.1 Motivation . . . 2
1.2 Project Goal . . . 3
1.3 Report Structure . . . 3

2 System Model and Notation **4**

2.1 Control Application Model . . . 4
2.2 (m, k) robustness requirement . . . 5
2.3 (m, k)-Pattern . . . 5
2.4 Schedulability and Scheduling . . . 6
2.5 Faults . . . 6
2.5.1 Causes of Transient Faults . . . 6
2.5.2 Impact of Transient Faults . . . 6
2.5.3 Faults in the Fault Tolerant Scheduler . . . 7

3 Implemented Compensation Techniques **8**

3.1 Static Pattern Based Execution . . . 8
3.2 Dynamic Compensation . . . 8
3.2.1 Postponing the (m,k)-Pattern . . . 9
3.2.2 Tolerance Counters . . . 10
3.2.3 Algorithm for Dynamic Compensation . . . 10
3.3 Summary of Soft-Error Handling techniques . . . 11

4 Fault Tolerant Scheduler integrated in RTEMS **13**

4.1 FTS Workflow . . . 13
4.2 Extending the RM Scheduler with the FTS . . . 14
4.2.1 Changes in *ratemoncreate.c* . . . 14
4.2.2 Changes in *ratemonperiod.c* . . . 15
4.3 Implementation of the Fault Tolerance Techniques . . . 15
4.3.1 SRE and SDR Pattern Iteration . . . 15
4.3.2 DRE and DDR Tolerance Counters . . . 15
4.4 FTS API . . . 17

4.4.1 Creating an RM FTS period 17
4.4.2 Fault Detection Routine 19
4.5 Fault Injection and Detection to test the FTS in RTEMS 20
4.5.1 Fault Injection Mechanism in RTEMS 20
4.5.2 Limitations of the Fault Injection and Error Handling 21
4.6 Example: How to use the FTS Scheduler 21
4.7 Enable the FTS in RTEMS 24

List of Figures **26**

Bibliography **28**

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 satellite Hitomi crashed, because its control loop got corrupted. According to investigations in [2] a bitflip occurred 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 catastrophes, 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 [13]. 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 is an option for space vehicles to manage redundant executions in the future.

1.3 Report Structure

The background of transient faults and the type of faults the Fault Tolerant Scheduler (FTS) can protect from is presented in the second chapter. The system model and notation is introduced in Chapter 3, including the task versions, the (m, k) robustness requirement, and the schedulability and scheduling. The soft-error handling techniques S-RE, S-DR, D-RE, and D-DR, which decide when to execute which task version, are the central theme in Chapter 4. Their theoretical background is covered briefly as well. In Chapter 5, the integration of the FTS in RTEMS, and tutorials on how to add and use the FTS are presented.

Chapter 2

System Model and Notation

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

2.1 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 [14], control flow checking [15], and majority voting [16]. We assume $c_i^u < c_i^d < c_i^c$ holds.

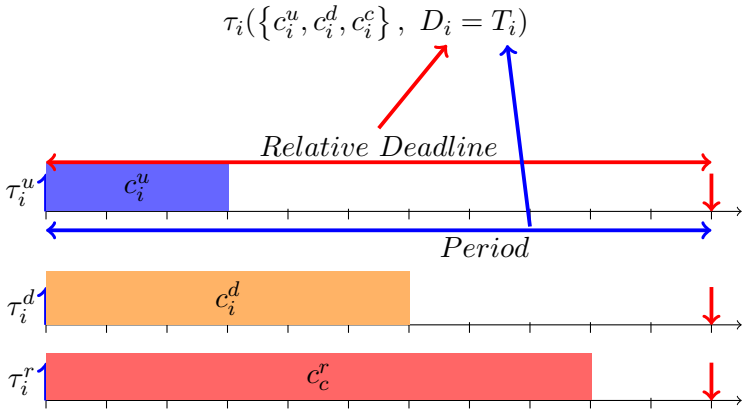

Figure 2.1: Possible setting of the task versions and their execution times, adapted from [17]

2.2 (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. 2.2 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.

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

2.3 (m, k)-Pattern

If only m out of k instances need to be correct, then, to guarantee the correctness, one way is to execute m of those task instances using τ_i^c. We will see later that there are other approaches to guarantee the correctness as well. To have an "execution plan" of when to execute which task version, the (m, k)-pattern [18, 19] 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 an unreliable instance if $\phi_{i,j} = 0$ and $\sum_{j=0}^{k_i-1} \phi_{i,j} = m_i$ [12].

For example, if the (m, k) robustness requirement is given as $(3, 5)$, then the (m, k)-pattern could be specified as $\Phi_i = \{0, 0, 1, 1, 1\}$.

There are two types of patterns available for the user, i.e., R-patterns and E-patterns [19], [18]. An R-pattern contains all "0"s at the beginning and all "1"s at the end, such as in $\{0, 0, 0, 1, 1, 1, 1\}$. An R-pattern is generated using the following rule:

$$\phi_{i,j} = \begin{cases} 1 & 1 \leq j \leq m_i \\ 0 & m_i < j \leq k_i \end{cases}$$
$$j = 0, 1, \ldots, k_i - 1$$

In E-patterns, the pattern begins with a "0", and the "1"s and "0"s are distributed evenly. It is created with the following formula:

$$\phi_{i,j} = \begin{cases} 0, & \text{if } j = \left\lfloor \left\lceil \frac{j \times (k_i - m_i)}{k_i} \right\rceil \times \frac{k_i}{(k_i - m_i)} \right\rfloor \\ 1, & \text{otherwise} \end{cases}$$

$$j = 0, 1, \ldots, k_i - 1$$

2.4 Schedulability and Scheduling

The FTS manages the executions of the different task versions by adopting Rate Monotonic 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].

2.5 Faults

In this section, transient faults and their impact on systems are introduced. An explanation of the application model, task versions, execution patterns, and the (m, k) requirement is included as well.

2.5.1 Causes of Transient Faults

Transient faults 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 under control.

2.5.2 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 cause system or mission failure. For example, 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.

Some faults may remain unnoticed, such as in case of silent data corruption. 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, which may lead to disasters.

2.5.3 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, cache, register, and during data transfer of these values between hardware components, such as sensor sampling data. We assume that the faulty value is within the range of the values the memory address or sensor value can adopt. For example, in case of a fault in sensor sampling data, the sensor sampling value can only assume values between the minimum and maximum possible sensor value. The system is protected by the FTS scheduler from incorrect calculations incurred by transient faults. When there is faulty control task output, it can be detected and corrected.

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, in that case the operating system may crash.

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 need to 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. This could lead to mission failure, such as in the case of satellite Hitomi [2].

Chapter 3

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 are discussed in this section.

3.1 Static Pattern Based Execution

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 3.1 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.

3.2 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 techniques. This chapter concludes the basics

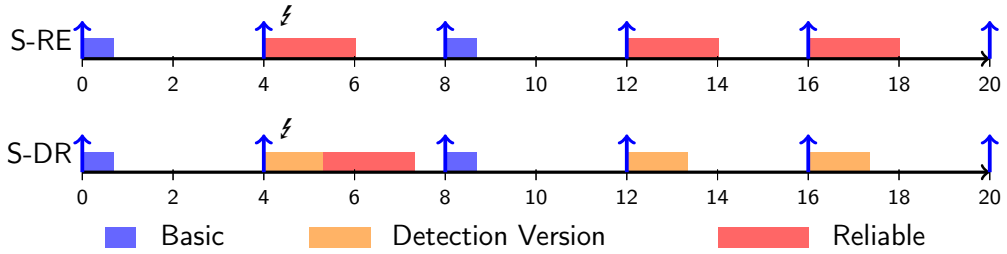

Figure 3.1: Examples for the static approaches for $(3, 5)$ with the pattern $\{0, 1, 0, 1, 1\}$, adapted from [17]

which are necessary to understand the techniques and the underlying principles of the FTS, 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 by executing at least m correct instances in a sliding window size of k. To detect errors in dynamic compensation, all instances that are executed before a "0", and become S if they are correct, need to be executed with τ^d.

3.2.1 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. As explained earlier, 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 task 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 $(3, 5)$, then 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, no error occurred; got away with it.*

3.2.2 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 "0"s in each partition and a_{ij} counts the number of "1"s in each partition. Consequently, the counter o_{ij} stands for the chances or tries in a partition the system has to be wrong, and a_{ij} stands for the number of instances which have to be correct after o_{ij} is depleted, when no more tries are allowed.

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 depleted by giving with tries with τ_i^d and failing due to errors, the system then has to follow the "1"s in 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 is specified. If the tolerance counter o_{ij} is depleted and the system is not allowed to give any more tries, since it would violate the (m_i, k_i) constraint in case of an error, Π will be set to safe; in this case the task has to be correct. However, if the task can still tolerate errors because the tolerance counter is not depleted, then Π is 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 tries with τ_i^d.

3.2.3 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 is set to safe mode. a_{ij} stores the number of instanced that have to be correct, 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, and o_{ij} and a_{ij} are replenished according to the new partition. When the system is in safe mode, there are two different strategies available. 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 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
```

Algorithm 3.1: Dynamic compensation of task τ_i with (m_i, k_i), adapted from [12]

3.3 Summary of Soft-Error Handling techniques

In this work, the four soft-error handling techniques are implemented. To give a quick overview, the techniques are presented one below the other.

In Figure 3.2, 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 is 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 violated 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.

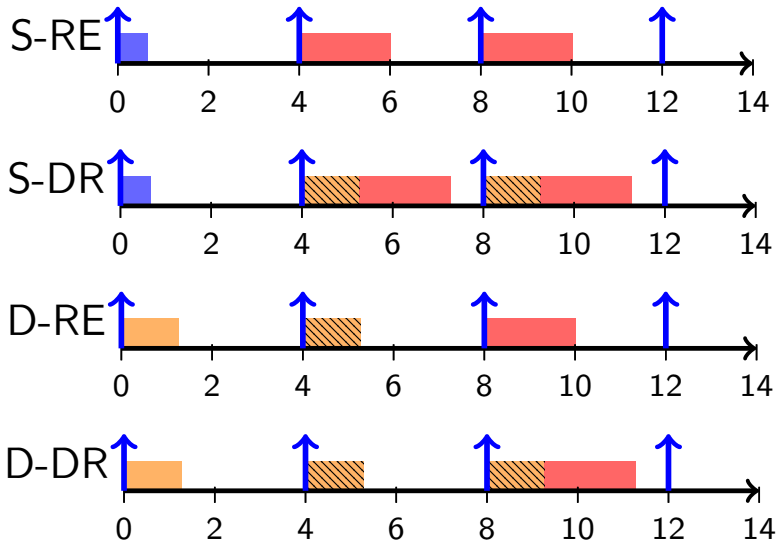

Figure 3.2: (m_i, k_i) is $(2, 3)$ and the static pattern is $\Phi = (0, 1, 1)$. Soft-errors occur in the second and third instance (marked with stripes). Blue block: τ_i^u, orange block: τ_i^d, red block: τ_i^c. Adapted from [12].

Chapter 4

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 [20] first. The rate monotonic (RM) scheduler was chosen to be extended with the FTS, because RM scheduling allows us to guarantee the schedulability of the system, and is widely used in practice. With a few modifications, the FTS can be used with other fixed priority scheduling algorithms too.

4.1 FTS Workflow

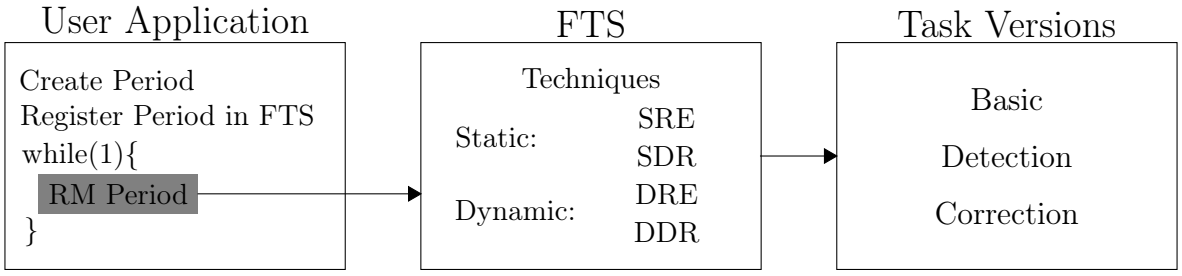

Figure 4.1: The flow diagram of the FTS. The while-loop calls RM Period, which in turn calls the FTS. The FTS, depending on its configuration, selects and releases the task versions.

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 [20], 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, as can be seen in Figure 4.1. The FTS then decides which version of the task to create and release. The

decision is made based on the configuration of the FTS, e.g., fault-tolerance technique, (m, k) requirement, and using R- or E-patterns.

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

4.2 Extending the RM Scheduler with the FTS

To integrate the FTS in the RM scheduling policy, the two files *ratemoncreate.c* and *ratemonperiod.c* were extended with function calls to the FTS. This section describes where in the RM scheduling code these function are called.

4.2.1 Changes in *ratemoncreate.c*

When the user usually creates a RM period using the function *rtems_rate_monotonic_create* (without using the FTS), a series of steps are executed. First, the RM scheduler allocates space for the period object that is created. Then, the system obtains a lock to protect the period object, to set its specifications in an exclusive manner. After setting the priority of the period, the period owner, which is the task initiating the creation of the period object, is stored in the period object in the next step, and the state of the period object is set to be inactive. Then, the watchdog timer is initialized and assigned to the period object. When the creation of the period object is completed, the period object ID is set and the lock can finally be released.

In the changed *ratemoncreate.c* file, after the object creation and the lock release, the function *fts_rtems_task_register_t*, which is implemented in the FTS, is called. With this function call, the period ID is registered for protection in the FTS. The code added in *ratemoncreate.c* can be viewed in the following.

```
1  /* Create period object and set its specification */
2    ...
3  /* Set FTS data */
4  uint8_t reg = fts_rtems_task_register_t(
5  id,
6  m,
7  k,
8  tech,
9  pattern_s,
10 versions,
11 specs
12 );
13
14 if ( reg == 0 )
15 {
16   printf("\nCould not register ID in the FTS\n");
17   return RTEMS_INVALID_NUMBER;
18 }
```

4.2.2 Changes in *ratemonperiod.c*

In *ratemonperiod.c*, two lines were added, which are call the FTS, which in turn release the task versions.

```
1 fts_version ver = fts_compensate_t(the_period->Object.id);
```

There are two places in the code where this function call to the FTS is made. The first insertion is in the function *_Rate_monotonic_Release_job*. This function is called when the RM scheduler wants to release a job. The owner of the period, which is the task which calls *rtems_rate_monotonic_period* to start a new period, is scheduled by the function call *_Scheduler_Release_job*. After the locks are released on the needed data structures to release a job of the owner, the function call *fts_compensate_t* is made, to call the FTS, which in turn starts the task versions based on its configuration.

The second insertion of *fts_compensate_t* is when postponed jobs are released, in the function *_Rate_monotonic_Release_postponed_job*, again at the end of this function when the a job of the owner is released, and all locks on data structured are released.

4.3 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 tolerance counters based on the (m, k)-pattern.

4.3.1 SRE and SDR Pattern Iteration

How an (m, k)-pattern can be specified with pointers in the FTS was described in the API section earlier. In the FTS, SRE and SDR follow this given static (m, k)-pattern to execute task versions while satisfying (m, k) requirements. The FTS iterates through this pattern with bit wise operations. The current byte's current bit is retrieved by bit wise 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.

4.3.2 DRE and DDR Tolerance Counters

At first, all tasks that were started in the previous period get deleted at the start of a new period. The comment in Line 5 explains why the task with the ID equal to zero must not be deleted. Then, the tolerance counters are processed, beginning in Line 32. 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 Algorithm 3.1. 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". After partitioning, for every partition in the (m, k)-pattern o_i and a_i are created. At the beginning of each period, to keep the tolerance counters up to date, the the RM period checks whether the tolerance counters are depleted. If they are depleted, they get replenished again.

The code which was added in *ratemonperiod.c* in the function *rtems_rate_monotonic_period* is shown below. The code is added at the beginning of this function, but after the lock for the period object has been acquired.

```
1  /* check if period is registered in the FTS */
2  int16_t i = task_in_list_t(the_period->Object.id);
3  if (i != -1)
4  {
5    /**
6     * At beginning of new period, delete all tasks from last period if their ID
     is not 0.
7     * If the ID is 0, which happens when the variables running_id_b/d/c are not
     set at all, or set to 0
8     * then don't delete the task, otherwise the task with ID==0 will be deleted,
9     * which is possibly the first task in the system.
10    *
11    * Delete all tasks at the beginning of a new period.
12    */
13   if (running_id_b[i] != 0)
14   {
15     task_status( rtems_task_delete( running_id_b[i] ) );
16     running_id_b[i] = 0;
17   }
18
19   if (running_id_d[i] != 0)
20   {
21     task_status( rtems_task_delete( running_id_d[i] ) );
22     running_id_d[i] = 0;
23   }
24
25   if (running_id_c[i] != 0)
26   {
27     task_status( rtems_task_delete( running_id_c[i] ) );
28     running_id_c[i] = 0;
29   }
30 }
31
32 /* If dynamic compensation is used, replenish tolerance counters when they are
     depleted */
33 if( (i != -1) && ( (list.tech[i] == DRE) || (list.tech[i] == DDR) )  )
34 {
35   if ( ((tol_counter_temp[i]).tol_counter_a[partition_index[i]]) == 0)
```

```
36      {
37        /* replenish only if at end of last partition (a is 0) */
38        if ( (nr_partitions[i] == partition_index[i]) && (tol_counter_temp[i].
      tol_counter_a[nr_partitions[i]] == 0) )
39        {
40          tolc_update(i);
41        }
42        else
43        {
44          /* was not in last partition */
45          partition_index[i]++;
46        }
47      }
48  }
```

4.4 FTS API

In this section, the API functions which the user has to call to use the FTS are presented. The maximum number of tasks that can be registered in the FTS can be changed in the header file *fts_t.h* with P_TASKS, and the maximum number of partitions of an (m, k)-pattern can be changed with PATTERN_PARTITIONS.

4.4.1 Creating an RM FTS period

The user has to call the function *rtems_rate_monotonic_create_fts*, which is implemented in *ratemoncreate.c*, to create a period and register it to the FTS for protection.

```
1  rtems_status_code rtems_rate_monotonic_create_fts(
2    rtems_name  name,
3    rtems_id   *id,
4    uint8_t m,
5    uint8_t k,
6    fts_tech tech,
7    pattern_specs *pattern_s,
8    task_versions *versions,
9    task_user_specs *specs
10 )
```

The *rtems_rate_monotonic_create_fts* has a lot of parameters, all of which have to be set. First, a name for the period has to be built, for example with:

```
rtems_name period_name = rtems_build_name('R', 'M', 'P', '');
```

```
rtems_id   *id
```

With *rtems_id* $*id$, the pointer to the ID of the RM period is passed.

```
uint8_t m
uint8_t k
```

With $uint8\_t\ \ m$ and $uint8\_t\ \ k$, m and k of the desired (m, k)-pattern are specified.

```
fts_tech tech
```

Using *tech*, the fault tolerance technique is chosen; available are SRE, SDR, DRE, and DDR, which are enums of fts_tech.

```
pattern_specs *pattern_s
```

The (m, k)-pattern needs to be specified using a *pattern_specs* struct, which contains a number of variables. First, the pattern type needs to be specified with either R_PATTERN or E_PATTERN. To be able to adopt the given (m, k)-pattern, the three parameters $uint8\_t*\ \ pattern\_start$, $uint8\_t*\ \ pattern\_end$, and $uint8\_t\ \ max\_bitpos$ have to be specified by the user[1]. The starting and ending address of the pattern is specified by *pattern_start* and *pattern_end*, which are pointers to a $uint8\_t$ variable.

```
typedef struct Pattern_Info {
  pattern_type pattern;
  uint8_t *pattern_start;
  uint8_t *pattern_end;
  uint8_t max_bitpos;
} pattern_specs;
```

Where the pattern should end, for example in midst of a byte, can be specified with the variable *max_bitpos* (set to 0 if the first bit of the last byte, and 7 if the last byte of the bit is chosen).

```
task_versions *versions
```

The struct holding the execution versions of a task is specified with the pointers to the three task versions basic b, detection d and correction c.

```
typedef struct Exec_Versions {
  rtems_task    *b;
  rtems_task    *d;
  rtems_task    *c;
} task_versions;
```

```
task_user_specs *specs
```

The specifications of the task versions when creation them in the FTS are specified with the following struct.

[1] Automatic allocation of memory space to store the (m, k)-pattern will be available later, after more tests with control applications have been done.

```
typedef struct Task_Specs_User {
  rtems_task_priority  initial_priority;
  size_t               stack_size;
  rtems_mode           initial_modes;
  rtems_attribute      attribute_set;
} task_user_specs;
```

When all the info for the FTS is ready, they can be passed to the RM scheduler, which in turn passes the info to register the period ID in the FTS.

4.4.2 Fault Detection Routine

At the end of the detection version, when the task finished computing, the user has to call the fault detection routine. The fault detection mechanism is implemented by the user in the detection version. The detection version calls the routine *fault_detection_routine* to report whether a fault occurred. The fault detection routine then initiates the reaction of the FTS to the fault. In case a fault is detected, the FTS can tolerate it, or release a correction task version when necessary.

The parameter list includes the ID of the period and the fault status of the computation of the task. The period ID is retrieved using the task argument of the detection version, see Line 11. When a fault was detected by the detection version, *fault_status* needs to be specified as FAULT, or in case no fault occurred, with NO_FAULT. The fault detection routine is called in Line 18.

```
1  void fault_detection_routine(
2    rtems_id id,
3    fault_status fs
4  )
5
6  /* Example for detection task version */
7  rtems_task DETECTION_V(
8    rtems_task_argument argument
9  )
10 {
11   rtems_id id = *((rtems_id*)argument);
12   printf("Detection: ID of my Period is %i\n", id);
13
14   /* Check whether a fault occurred */
15   fault status fs_T1 = get_fault(get_rand());
16
17   /* Call fault detection routine at the end of the detection version activity */
18   fault_detection_routine(id, fs_T1);
19   rtems_task_delete( rtems_task_self() );
20 }
```

4.5 Fault Injection and Detection to test the FTS in RTEMS

4.5.1 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 in Line 1.

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 $get\_rand()$ in Line 10, and the value $rand\_count$ specifies which random value has already been used. The function $get\_fault(...)$ in Line 15 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.

```
1  void rand_nr_list(void)
2  {
3    for ( uint16_t i = 0; i < NR_RANDS; i++ )
4    {
5      rands_0_100[i] = rand() / (RAND_MAX / (100 + 1) + 1);
6    }
7    return;
8  }
9
10 uint8_t get_rand(void)
11 {
12   return rands_0_100[rand_count++];
13 }
14
15 fault_status get_fault(uint8_t rand_nr)
16 {
17   if ( fault_rate == 0 )
18   {
19     return NO_FAULT;
20   }
21   else if ( rand_nr <= fault_rate )
22   {
23     faults++;
24     return FAULT;
```

```
25   }
26   else
27   {
28     return NO_FAULT;
29   }
30 }
```

4.5.2 Limitations of the Fault Injection and Error Handling

To demonstrate that the FTS works correctly, an event is used to simulate the occurrence of a fault. The fault makes the FTS perform error handling routines, such as tolerating the error or changing to a correction version of the task. Error detection and correction are also assumed to be perfect, which means that we can always recover from soft-errors. This approach is a preliminary one, but we hope to improve the simulation of faults, as well as the error handling on other hardware in the future, for example on smaller robots.

4.6 Example: How to use the FTS Scheduler

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

```
1  /* Basic task version */
2  rtems_task BASIC_V(
3    rtems_task_argument argument
4  )
5  {
6    rtems_id id = *((rtems_id*)argument);
7    printf("Basic: ID of my Period is %i\n", id);
8    rtems_task_delete( rtems_task_self() );
9  }
10
11 /* Detection task version */
12 rtems_task DETECTION_V(
13   rtems_task_argument argument
14 )
15 {
16   rtems_id id = *((rtems_id*)argument);
17   printf("Detection: ID of my Period is %i\n", id);
18
19   /* Check whether a fault occurred */
20   fault status fs_T1 = get_fault(get_rand());
21
22   /* Call fault detection routine at the end of the detection version activity */
```

```
23   fault_detection_routine(id, fs_T1);
24   rtems_task_delete( rtems_task_self() );
25 }
26
27 /* Correction task version */
28 rtems_task CORRECTION_V(
29   rtems_task_argument argument
30 )
31 {
32   rtems_id id = *((rtems_id*)argument);
33   printf("Correction: ID of my Period is %i\n", id);
34   rtems_task_delete( rtems_task_self() );
35 }
```

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. From Lines 1-11 in the code below, the (m, k)-pattern specifications are initialized. The structs for the task versions and the user specifications for the task versions are initialized in Lines 14 and 17. In Line 20 and 21, the (m, k) requirement is specified, and in Line 21, the fault tolerance technique is set.

```
1  /* last byte of (m,k)-pattern */
2  uint8_t end_p = 0;
3  /* first byte of (m,k)-pattern */
4  uint8_t begin_p = 0;
5  /* store addresses */
6  uint8_t * p_s = &begin_p;
7  uint8_t * p_e  = &end_p;
8  /* pattern specifications */
9  pattern_specs p_specs;
10 pattern_type patt = E_PATTERN;
11 uint8_t maxbit = 7;
12
13 /* task versions struct containing the task pointers to the execution versions */
14 task_versions versions;
15
16 /* task user specifications */
17 task_user_specs task_sp;
18
19 /* set (m,k) */
20 uint8_t m = 12;
21 uint8_t k = 16;
22
23 /* set technique */
24 fts_tech curr_tech = DDR;
```

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. In Lines 6-10, the variables for the period, a status code variable, and an ID is initialized. In

Lines 13-15, the task entry points of the task versions are stores in a struct. In Lines 18-22, the specifications of the task versions are set, and stored in a struct. The period is finally created in Lines 31 and 32, by calling *rtems_rate_monotonic_create_fts*. The while loop in Line 34 calls *rtems_rate_monotonic_period* in every invocation. This calls the FTS, which creates and starts the task versions.

```
1  /* Task 1 contains the while-loop */
2  rtems_task FTS_TEST(
3    rtems_task_argument unused
4  )
5  {
6    rtems_status_code status;
7    rtems_id RM_period;
8    rtems_name rm_name = rtems_build_name( 'R', 'M', 'P', ' ');
9    rtems_id selfid = rtems_task_self();
10   rtems_rate_monotonic_period_status period_status;
11
12   /* Store addresses of task pointers in struct */
13   versions.b = &BASIC_V;
14   versions.d = &DETECTION_V;
15   versions.c = &CORRECTION_V;
16
17   /* Store pattern specifications in struct */
18   p_specs.pattern = patt;
19   p_specs.pattern_start = p_s;
20   p_specs.pattern_end = p_e;
21   p_specs.max_bitpos = maxbit;
22
23   /* All task versions have the same specifications, store them in struct */
24   rtems_task_priority prio = 1;
25   task_sp.initial_priority = prio;
26   task_sp.stack_size = RTEMS_MINIMUM_STACK_SIZE;
27   task_sp.initial_modes = RTEMS_DEFAULT_MODES;
28   task_sp.attribute_set = RTEMS_DEFAULT_ATTRIBUTES;
29
30   /* create a period and register the task set for RM-FTS */
31   status = rtems_rate_monotonic_create_fts( rm_name, &RM_period,
32    m, k, curr_tech, &p_specs, &versions, &task_sp );
33
34   while (1)
35   { /* task versions are released by the FTS in every new period */
36     status = rtems_rate_monotonic_period( RM_period, 100 );
37
38     if( status == RTEMS_TIMEOUT )
39     {
40       printf("\nPERIOD TIMEOUT\n");
41       break;
42     }
43   }
```

```
44
45   status = rtems_rate_monotonic_delete( RM_period );
46   if ( status != RTEMS_SUCCESSFUL )
47   {
48       printf( "rtems_rate_monotonic_delete failed with status of %d.\n", status );
49       exit( 1 );
50   }
51   status = rtems_task_delete( selfid );     /* should not return */
52   printf( "rtems_task_delete returned with status of %d.\n", status );
53   exit( 1 );
54 };
```

4.7 Enable the FTS in RTEMS

The requirement for adding the FTS to an RTEMS system is that the RTEMS tools and kernel were already built, for example by following the Quick Start guide in the RTEMS user manual [21]. To add the FTS to an RTEMS system, navigate to the kernel, then into *cpukit/rtems*, open *Makefile.am*, and add the following lines:

```
1 include_rtems_rtems_HEADERS += include/rtems/rtems/fts_t.h
2 librtems_a_SOURCES += src/fts_t.c
```

The *.h* line belong where the other header files are included too, but above *if HAS_MP*. The *.c* line has to be added at the bottom of the file, but above *if HAS_MP* as well.

Then navigate to the location of the classical C API header files and copy the *fts_t.h* file there (in *cpukit/rtems/include/rtems/rtems* for me). The *fts_t.c* file needs to be copied into the source folder of the classical C API (in *cpukit/rtems/src* for me). Next, replace the standard *ratemoncreate.c* and *ratemonperiod.c* files with the ones from my repository. There will be a package with all the files needed, with the name "rmftspatch" and uploaded in [22].

From here on, all the file locations for the commands below need to be adapted to your own system. The next step is to bootstrap. Make sure to set your path with

```
1 export PATH=$HOME/development/rtems/4.12/bin:$PATH
```

and navigate to the location of bootstrap, *rtems/kernel/rtems*. The bootstrapping can proceed:

```
1 ./bootstrap -c && ./bootstrap -p && \
2 $HOME/development/rtems/rsb/source-builder/sb-bootstrap
```

We then navigate to */kernel/erc*32 and run the configure script, for the erc32 board support package, the command is:

```
1 $HOME/development/rtems/kernel/rtems/configure --prefix=$HOME/development/rtems
      /4.12 \
2 --target=sparc-rtems4.12 --enable-rtemsbsp=erc32 --enable-posix
```

The only thing left to do is running

```
1 make
2 make install
```

to compile the system. You can now access the FTS code in your user application by adding

```
1 #include <rtems/rtems/fts_t.h>
```

in your implementation.

List of Figures

2.1 Possible setting of the task versions and their execution times, adapted from [17] 4
2.2 The bitmap of a task τ_i for eight executed instances. "1" represents instances without a fault, and "0" for instances with a fault. 5

3.1 Examples for the static approaches for $(3, 5)$ with the pattern $\{0, 1, 0, 1, 1\}$, adapted from [17] 9
3.2 (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]. 12

4.1 The flow diagram of the FTS. The while-loop calls RM Period, which in turn calls the FTS. The FTS, depending on its configuration, selects and releases the task versions. 13

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] RTEMS Website. `https://www.rtems.org/`, 2017.

[14] 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.

[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] 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.

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

[18] 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.

[19] 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.

[20] Rate Monotonic Manager. `ftp://ftp.rtems.org/pub/rtems/people/chrisj/docs/c-user/rate_monotonic_manager.html`, 2017.

[21] RTEMS User Manual. `https://docs.rtems.org/branches/master/user.pdf`, 2017.

[22] Github Repository, Mikail Yayla. `https://github.com/myay/rtems_faulttolerance_SOCIS17`, 2017.