Special Session: Operating Systems under test: an overview of the significance of the operating system in the resiliency of the computing continuum

Emmanuel Casseau, Petr Dobiáš
Univ Rennes, Inria, CNRS, IRISA
Lannion, France
petr.dobias@sorbonne-universite.fr
emmanuel.casseau@irisa.fr

Oliver Sinnen
Dept. of Electrical, Computer and Software Engineering
University of Auckland
Auckland, New Zealand
o.sinnen@auckland.ac.nz

Gennaro S. Rodrigues, Fernanda Kastensmidt
Instituto de Informatica, PGMicro
UFRGS
Porto Alegre, Brazil
{gsrodrigues, fglima}@inf.ufrgs.br

Alessandro Savino, Stefano Di Carlo, Maurizio Rebaudengo
Control and Computer Engineering Department
Politenico di Torino
Torino, Italy
{name.surname}@polito.it

Alberto Bosio
Univ Lyon, ECL
INL, UMR5270
Ecully, France
alberto.bosio@ec-lyon.fr

2021 IEEE 39th VLSI Test Symposium (VTS) | 978-1-6654-1949-9/20/$31.00 ©2021 IEEE | DOI: 10.1109/VTS50974.2021.9441042

***Abstract*—The computing continuum's actual trend is facing a growth in terms of devices with any degree of computational capability. Those devices may or may not include a full-stack, including the Operating System layer and the Application layer, or just facing pure bare-metal solutions. In either case, the reliability of the full system stack has to be guaranteed. It is crucial to provide data regarding the impact of faults at all system stack levels and potential hardening solutions to design highly resilient systems. While most of the work usually concentrates on the application reliability, the special session aims to provide a deep comprehension of the impact on the reliability of an embedded system when faults in the hardware substrate of the system stack surface at the Operating System layer. For this reason, we will cover a comparison from an application perspective when hardware faults happen in bare metal vs. real-time OS vs. general-purpose OS. Then we will go deeper within a FreeRTOS to evaluate the contribution of all parts of the OS. Eventually, the Special Session will propose some hardening techniques at the Operating System level by exploiting the scheduling capabilities.**

***Index Terms*—Operating Systems, RTOS, Reliability, Fault Tolerance, Fault Injection, Task Scheduling**

I. Introduction

Nowadays, embedded systems are employed in several fields, spanning from consumer electronics (e.g., mobile phones) to safety-critical applications such as automotive, aerospace, and avionic. To meet the application constraints escalation, embedded systems' computing capabilities have increased over the years, as object detection on autonomous driving can manifest [1]. For these reasons, embedded systems are growing in complexity, including multi-core systems, often characterized by integrating more than one Central Processing Unit (CPU) and Graphical Processing Unit (GPU) on the same chip. As a direct consequence, parallel programming paradigms were introduced, significantly improving the computational throughput. In order to cope with such computational complexity, the full system stack requires avoiding bare-metal solutions. Instead, it needs a middle layer to deploy the final application properly. The middle layer is commonly composed of the Operating System (OS) and the middleware (e.g., peripheral drivers), delivering mechanisms such as synchronization primitives, i.e., semaphores, mutexes, as well as asynchronous I/O.

Despite all innovations, according to their *mission*, embedded systems must still meet a set of both required and desirable features, chosen at design time and imposed by standards; *dependability* is usually among them. Indeed, dependability reduces in many ways: without considering errors due to design, hardware, or software bugs, issues that occurred during fabrication, intentional tampering, or any other external events that can affect this property during the lifetime of the application. Some of those events are due to the circuit's interaction with the surrounding environment, causing problems like memory bit-flip, signal degradation, data loss, permanent damage to the physical circuit [2].

Usually, an in-deep analysis of the system, targeting its weaknesses, allows the system to achieve the system's dependability and then implement mitigation techniques that reduce or altogether remove such weakness. However, extensive testing phases come at the cost of money and time, generally delaying the time-to-market and increasing the final per-unit price. For those reasons, the design phase must find an optimal trade-off so that the product's final price does not exceed the target one and, at the same time, the system can still work with the desired quality level. When literature considers the full system, the reliability analysis target is commonly

978-1-6654-1949-9/21/$31.00 ©2021 IEEE

the application-layer [3]. However, safety-critical systems may need to manage the execution of many applications, sharing resources. To guarantee safe management of those resources, using an operating system (OS) is attractive when running bare metal applications on a system could lead to a waste of resources. Therefore, it is crucial to evaluate the possibility of OS usage and its effects on system behavior and fault tolerance. Some works [4] already proved that an application's fault tolerance might differ a lot when executing bare metal or on top of a complex operating system such as Linux. Thus, it is mandatory to study the most critical OS data structures and eventually propose a specific fault-tolerant mechanism for OS.

This paper presents an overview of the interplay between the full system reliability and the operating system's presence. The overview includes evaluating several applications under fault injection simulation, executing both on bare metal systems and on top of FreeRTOS and Linux. The evaluation proved that a lighter OS such as FreeRTOS would have less impact on the system susceptibility to errors than a more robust, complex ones such as Linux. Then, the paper move to the study of the reliability of FreeRTOS affected by Single Event Upset (SEU) faults. The applied methodology targets all the most relevant variables and data structures of FreeRTOS analyzed through a fault injection campaign. Using the FreeRTOS operating system as a case study, the paper evaluates the impact on the application in terms of system integrity, data integrity, and the overall resistance to faults. The last part of the paper is devoted to hardening techniques at the OS level. More in detail, we discuss the analysis of the impact of task scheduling when a multiprocessor system can make use of redundancy in both time and space. Since offline scheduling is not appropriate for OSs that need to react immediately when a new task arrives or when a fault occurs, the paper suggests an online scheduling approach to deal in real-time.

The remainder of this paper is structured as follows. Section II presents the basics knowledge and state of the art. Section III details the Operating System's impact on the full system reliability. In contrast, Section IV explores the most critical resources of a Real-Time Operating System to the full system reliability. In Section V proposes a fault-tolerant technique for the Operating System, and finally Section VI draws conclusions and perspectives on the matter.

II. Background

This section presents the state-of-the-art about *Fault Injection* techniques (including a discussion about related works) and the basics knowledge regarding *Real-Time Operating System*.

A. Fault Model

A *Single Event Effect* (SEE) is the electrical noise induced in a circuit by a natural phenomenon external to the circuit itself. SEEs are caused by *ionizing particles* that collide with electronic devices: they may come from deep space (cosmic and gamma rays), Sun (solar wind), magnetosphere (van Allen belts), and Earth's crust (from naturally radioactive materials). Those ionizing particles can be massive (heavy ions, protons, neutrons, electrons) or mass-less (photons). Furthermore, the incriminated particles causing SEE can directly impact the device or create a secondary cascade of particles when entering the atmosphere.

SEEs are dangerous because they may alter the technology behavior, and their incidence becomes even more prominent because of the continuous scaling of technology. Their severity can be analyzed considering the consequences of their impact on the circuit and how it harms its mission. SEE's consequences and the probability that a phenomenon has visible effects on a circuit can be estimated by reproducing the phenomenon with testing techniques. With these tests' results, it is possible to know which are the most sensitive parts of the system, classify the misbehavior, and, eventually, develop new methods to solve, or at least reduce, some vulnerabilities.

Notoriously, on-board electronics used in avionics and aerospace are the most subjected to SEE because they work at high altitudes or even outside the atmosphere, where there is no protection against these phenomena. Nevertheless, many sudden failures in electronic devices working ashore (consumer electronics, industrial applications, automotive circuits) are the results of ionizing particles reaching the ground, [5].

In general, the damage caused by SEE classifies them, together with the harming mechanism they induce. JEDEC standards [6] identify several classes of SEEs, while in the following, only the most relevant ones are considered:

- **Single Event Upset (SEU)**: This is one of the most studied SEEs because it is very common [7]: caused by the interaction of a particle with a memory element, it induces a change of state (known as *bit-flip* too). The most important SEU types are Single Bit Upset (SBU) and Multiple Bit Upset (MBU). It is still considered critical as it may lead to the modification of a memory cell's content, which will be random in time and location. The randomness of the event usually relates to unexpected consequences for the application level.
- **Single Event Transient (SET)**: A momentary voltage spike causes this event in a precise position of a circuit, originated by a sudden event like a high energy particle hitting the device or a strong electromagnetic interference with a near source. If a memory element retains such a transient value of the signal, misbehavior could be seen.

Thus, we can model the SEE as a memory cell logic flip (from '0' to '1' or *vice versa*) since it can be due directly to an SEU or because the memory cell stored a wrong value due to a SET.

Figure 1 sketches how SEEs reach and manifest themselves at the software layer by impacting both OS and running applications. A straightforward way to model faults is to map them into a set of fault models that affect either the instructions or the software layer's data. Examples of software-level fault models are the Wrong Data in Operand (WDat) and the Instruction Replacement (InstR) [8]–[11]. They all model the effect of transient/permanent faults occurring either in the

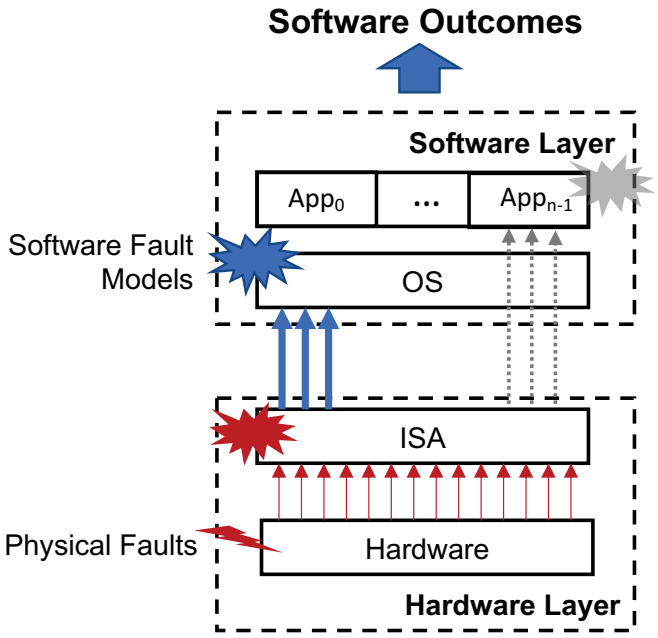

Fig. 1: Fault Propagation through System Layers

memory segment storing the program's data (WDat) or in the memory segment storing the code (InstR). In this work, we focus on the Wrong Data in Operand fault model since we are interested in understanding the impact of SEE in the OS data structure.

B. Fault Injection

Fault injection is very well-known and a useful technique to evaluate the reliability of the systems under faults [12]–[15]. It is based on the realization of controlled experiments to observe the system's performance in the presence of faults. Fault injection can be classified based on the mechanisms used to introduce the fault into the system artificially: (i) Physical Fault Injection aims at exposing the final implementation of the system to an external fault source (e.g., radiation-based fault injections), while (ii) Hardware-based fault injection exploits existing hardware interfaces to alter the behavior of the system (e.g., using debug interface to access to CPU internal registers), (iii) Software-based fault injection instruments the software layer of the system in order to inject faults directly into memory locations, and (iv) model-based fault injection instruments the system's model (e.g., HDL model) to be able to inject faults during model simulation/emulation. In [15] the reader can find examples and details of the above techniques.

Every injection could lead to different types of behavior of the system classified as follows:

- **OK:** the system continues working expectedly, without showing any appreciable difference concerning the golden run after the injection;
- **Crash**: When a critical error occurs, and the system stops working (or reset itself), we are in the presence of a *crash*.
- **Freeze**: A misbehavior is classified as *freeze* when all running tasks continue to run after the supposed deadline, and they still do not end after a time window of observation. This time window can be defined depending on the system requirements.
- **Silent Data Corruption (SDC)**: A misbehavior is classified as *SDC* when, despite the application being able to end its run, the output of the computation contains any deviation from the golden one.
- **Delays**: Whenever a task completes its duty (without SDC) but the execution time exceeds the expected one, the difference is a missed deadline; in this case, the misbehavior is classified as a *Delay*.

C. Hardening

To make system *fault tolerant*, i.e. more *robust* against faults, one of commonly used techniques is redundancy. *Redundancy* is the provision of functional capabilities that would not be necessary for a fault-free environment [16], [17]. It can be in time or space. *Time redundancy* consists of repeating the same computation or data transmission to make a comparison later and check for faults. *Space redundancy* can be classified into three types depending on the type of redundant resources added to the system [16], [18], [19]. *Hardware redundancy* makes use of additional components, such as processors or memories. *Software redundancy* considers that (i) a function to improve system fault tolerance is added to an already existing code or (ii) several versions of one function are codded, and results are compared. *Information redundancy* takes advantage of coding by adding a piece of supplementary information, e.g., Hamming codes or cyclic redundancy check.

Although the redundancy improves the system's reliability, its overheads are not negligible. While it is not possible to prevent the overheads due to space redundancy, the ones caused by time redundancy can be avoided. If, after the first execution, no fault is detected, a new execution is not necessary.

III. Evaluating the Impact of Operating System on the Application Behavior under Soft Errors

The goal of this evaluation is to determine the impact of the OS on application resilience. The following subsections will present the experimental conditions, simulator, CPU architecture, Fault Injection approach, and applications under test. The last one will discuss the obtained results.

As explained in Section II, the OS evaluation consists of comparing a golden execution of the application (i.e., with no-fault injection) and the executions under fault injection. Three versions of the same application are included in the evaluation: a bare-metal implementation, thus executed without OS, a version running on top of the FreeRTOS operating system, and a version running on top of Linux OS.

A. Simulator

The OVPSim simulator [20] is the tool used to support the experimental evaluation. OVPSim is a full-system simulator used to simulate the execution of a code in the target

hardware. It uses just-in-time binary translation, achieving high simulation speeds. That makes OVP an instruction-accurate simulator, providing the possibility to analyze each instruction's execution, but not real execution times.

B. ARM Cortex-A9 Model

The target hardware is the ARM Cortex-A9, with one processor core. The recurrent presence on safety-critical applications based on commercial off-the-shelf (COTS), such as the Xilinx Zync-7000 platform, makes this CPU a perfect candidate. The model used to simulate the ARM Cortex-A9 was the one developed by ARM Ltd.

C. Fault Injection

The analysis compares an error-free run (called golden execution) of the system and the faulty runs under the effects of fault injections. The OVPSim-FIM (OVP Fault Injection Module) [21] was used in this work for fault injection and error evaluation. Only the processor registers are targets of this investigation.

D. Applications

We resort to two main sets of benchmarks: (i) Successive Approximation (SA) algorithms and (ii) General Purpose (GP) algorithms. The two sets have a different intrinsic resilience to SEE. SAs are more resilient than GPs because of their inherent redundancy [22]. We selected those two sets to have different resiliency w.r.t. SEE and show that the OS will have a significant role independently on the executed application. More in detail, we have three different SAs: Newton-Raphson and Trapezoid, both useful to compute the integral of a function and the QSolver for root computation of quadratic equations. The GPs set comprises four benchmarks: Matrix Multiplication, Vector Sum, and Tower of Hanoi puzzle solver benchmarks.

E. Results and Discussion

Table I presents each application and execution results, divided by every raised error type. It is crucial to highlight that the exception errors include segmentation faults and every other (unidentified) exception.

F. Bare Metal

The successive approximation algorithms are much less susceptible to SDC errors than the three other applications. Comparing the worst-case scenario for the successive approximation algorithms (Newton-Raphson) with the other three (Hanoi) best-case scenarios, we have that the former is about 26% less susceptible to SDC errors than the latter. With the best-case scenario for the successive approximation (QSolver) compared with the worst case from the other three (Vector Sum), we find that the former may be up to 96% less susceptible to SDC errors than the latter. That means successive approximation algorithms may be from 26% up to 96% more reliable from SDC errors than ordinary calculation algorithms when executing bare-metal. On the other hand, according to Table I, those algorithms are more susceptible to Freeze errors. Here the main point is not discussing which are the most/less resilient to SEE but to compare the SDC rates w.r.t. of executing the same application on the top of an OS. From Table I, we can observe that executing bare metal applications have a higher percentage of OK, which means they generated much fewer errors than Linux or FreeRTOS.

G. FreeRTOS

Comparing the worst-case scenario for the successive approximation algorithms (Trapezoid) with the best-case scenario of the other three (Vector Sum), we have that the former are about 28% less susceptible to SDC errors than the latter. With the best-case scenario for the successive approximation (Newton-Raphson) compared with the worst case from the other three (Hanoi), we find that the former may be up to 78% less susceptible to SDC errors than the latter. With that data, we see that, on our tests, successive approximation algorithms are from 28% to 78% less susceptible to SDC errors than ordinary calculation algorithms when executing on top of FreeRTOS. We see that FreeRTOS applications are much more susceptible to hangs than their Linux and bare metal counterparts. An application's distribution of errors differs when executing bare metal or on top of an operating system.

H. Linux

The successive approximation algorithms did not have better fault tolerance than the typical computing applications when running on Linux. However, the successive approximation algorithms maintained a better susceptibility to SDC errors on average.

As already seen in past works [4], it is clear that the usage of an operating system drastically changes the fault tolerance. That is because the OS itself is a target to faults that may cause errors. In our simulations, we inject one fault per execution. Therefore, a significant application will have a higher probability of having a fault injected during its execution. On the contrary, there are higher chances for a small application that the fault will happen (hence injected) during some OS function execution [23]. The usage of successive approximation algorithms has its natural fault tolerance masked because of the OS criticality.

It can also be observed that Linux executions have to deal with segmentation fault errors, which are not present on FreeRTOS and bare metal. Nevertheless, those exceptions represent errors caught by the operating system. In the absence of an operating system, those errors could manifest themselves as other types of errors.

As expected by theoretical analysis, numerical methods have natural inherent fault tolerance because of their iterative nature. Given this nature, a fault injected during one iteration may be masked on the next ones unless it causes a permanent hardware error.

The question we then further investigate is what are the most sensitive OS data structures to SEE. If identified, this investigation will be mandatory to provide a fault tolerance mechanism at the OS level.

TABLE I: Error Distribution

Execution		Errors [%]				
		General			Exceptions	
OS	Application	OK	SDC	Freeze (+ Crashes)	Segmentation Fault	Unidentified
Bare Metal	QSolver	82.1	0.9	17.0	-	-
	NewtonRaphson	77.1	9.6	13.3	-	-
	Trapezoid	87.2	3.4	9.4	-	-
	Matrix Multiplication	70.1	19.5	10.4	-	-
	Vector Sum	65.2	23.6	11.2	-	-
	Hanoi	77.8	13.0	9.2	-	-
FreeRTOS	QSolver	43.0	9.6	47.1	-	0.4
	Newton Raphson	74.5	8.7	15.9	-	0.9
	Trapezoid	58.3	10.4	31.2	-	0.1
	Matrix Multiplication	42.1	16.6	40.9	-	0.4
	Vector Sum	42.3	14.6	43.0	-	0.1
	Hanoi	24.5	40.8	30.7	-	4.0
Linux	QSolver	53.5	19.4	9.3	6.7	11.1
	Newton Raphson	53.5	18.8	9.4	6.0	12.2
	Trapezoid	55.7	28.5	0.3	15.3	0.2
	Matrix Multiplication	45.4	37.0	4.8	12.3	0.5
	Vector Sum	43.6	40.2	4.9	10.9	0.5
	Hanoi	58.1	17.5	7.8	8.1	8.5

IV. Data Structures role on the reliability of a Real Time Operating System

Analyzing the most sensitive OS data structures requires a proper Environment. As presented in [24], using specific Fault Injection Environment (FIE) it is possible to target the fault injection of all data structures included in the FreeRTOS OS [25]. In general, RTOSs can schedule concurrent operations belonging to different contexts in the form of *tasks* (or *processes*) and switch among them in such a way that desired timing should always be respected. Each task can be in a defined state at every moment of its lifetime, and the programmer can partially choose how and when a task must change it. All operating systems (not only RTOSs) commonly recognize at least three states for each task: (i) the *ready* state, when the task can be scheduled at any time, (ii) the *running* state when the task has been switched in and runs, holding the core of the processor, and (iii) the *waiting* state when the task is waiting for an event to happen. Sometimes OS supports two additional states: (i) the *new* state, to identify newly (hence never scheduled) created task, and (ii) the *deleted* state, to define to-be-removed tasks. Deleted tasks are waiting for the kernel to clear their stack and free all memory locations associated with those tasks.

The FIE fully described in [24] is composed of a board that acts as DUT and by a Host machine working as the platform that configures and controls the injection campaign and logs all results. From a generic point of view, the host-side program sends the DUT a sequence of injection parameters, such as the injection target, the injection time, when the application starts. Then, the DUT is left free to run for a defined amount of time, after which an asynchronous interrupt routine injects the desired target. The DUT uses a resume routine to send back to the host results of the injection.

All investigations use the STM32F3DISCOVERY (STMicroelectronics®) board as DUT. The board features an STM32F303VCT6 micro-controller based on the ARM® Cortex® M4 Architecture, working at 72MHz, along with an ST-LINKv2 debugger interface, while the Host machine is a Linux-based operating system. Moreover, the board is fully compatible with the FreeRTOS. Figure 2 shows a scheme of the whole system.

This investigation follows the behavior classification explained in Section II since every injection may lead to different types of behavior of the system:

- **OK:** the system continues working expectedly, without showing any appreciable difference to the golden run after the injection. The golden run is the first run executed on the DUT without any injection at the beginning of each fault injection campaign.
- **Crash**: When a critical error occurs on the DUT, the internal reset handler fires to avoid further problems, signaling a *crash*.
- **Freeze**: A misbehavior is classified as *freeze* when all running tasks continue to run after the supposed deadline, and they still do not end after a time window of observation. This time window can be defined depending on the system requirements. Reported experimental results refer to a dynamic time window defined as 50% more of the longer task's expected timing.
- **Silent Data Corruption (SDC)**: A misbehavior is classified as *SDC* when, despite the application being able to end its run, the Host detected an alteration in the output

of the computation. A CRC signature of the execution's output values allows for comparing the golden execution and the fault one when SDC comes into play.

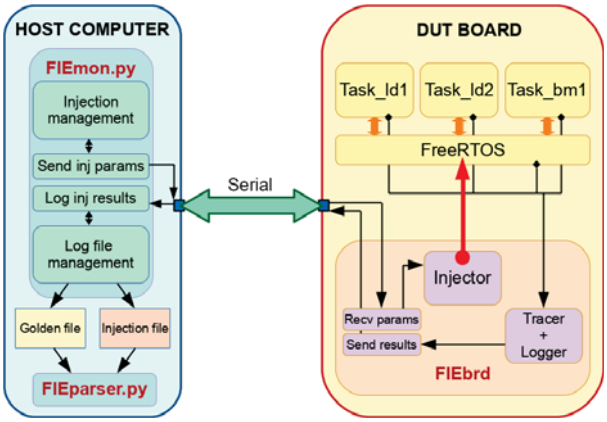

Fig. 2: Fault Injection Environment Model

The OS maintains several data, and the analysis capability allows to target them selectively, among them we studied:

- **Tasks list**: The state of a task is determined based on the state list where the task belongs. Injections target both the ready task list (*RDYLST*) and the delayed task list (*DLDLST*). The same structure defines both lists, including five fault locations within it.
- **Mutex and Queue structure** (*MTXQVARS*): The queue is a structure that can be used as a semaphore or mutex by the kernel. It is composed of 22 fault locations.

Among the EEMBC® Automotive suite [26], we selected few benchmarks to include as tasks running on the FreeRTOS, evaluating the impact of its data structures vulnerability to SEU. This way, we ensure future comparisons with the results by performing experiments using standard programs. The selection includes a set of benchmarks that reproduces some very common calculations in the automotive field. While these benchmarks target multi-core processors to test the used platform's scalability, they have been used in a single-core microcontroller because the performance analysis is not relevant in this work. While the full set of benchmarks comprises 16 applications, the tests include three benchmarks:

- `a2time` (Angle to time conversion): this benchmark simulates an engine with different cylinders (4, 6, or 8, to be chosen before compilation) with a crankshaft, a toothed reluctor wheel, and a sensor able to generate a pulse every time it detects the passage of a tooth: this type of mechanism controls the fuel injection in the cylinders and the subsequent spark.
- `tblook` (Table lookup and interpolation): a table lookup algorithm to store a limited amount of data pairs, coming from one or more resources (sensors, connections, calculations), and to interpolate missing pairs. It is commonly used in embedded systems when the memory resources are limited, and only portions of data can be stored.
- `idctrn` (Inverse Discrete Cosine Transform): the implementation of the Inverse Discrete Cosine Transform (IDCT) widely used in digital graphics; the actual implementation applies the IDCT to an input data-set representing a matrix of 64 bits values. It is the only multi-thread benchmark considered in this work.

A. Results Analysis

In general, experimental results confirm that FreeRTOS is affected by some vulnerabilities. As expected, most critical vulnerabilities are pointers and numerical indexes stored in integer variables (both signed and unsigned) to address elements of lists or vectors.

Figure 3 reports the classification of the fault injection over the three target location groups. The most sensible group to faults is the ready list group (RDYLST), which almost always leads to crashes because it includes all data required to schedule alive threads properly. The other thread list (DLDLST) goes into an idle state, i.e., crash or freeze, in less than 50% of the injected faults, while most of the time, an SDC is the result of the computation, making the OS more resilient than expected. Eventually, the MTXQVARS group shows a prevalence of SDCs among other classes and less than 30% of crashes and freezes as expected since they include locations controlling mutex and semaphores.

All those considerations point out that FreeRTOS can gain from its hardening in different ways. For example, by merely duplicating or triplicating the most sensitive data, the reliability will be guaranteed, but results already show that a full duplication or triplication would be excessive. Moreover, the voting systems will add some computational overhead to all kernel procedures, which might disrupt the OS's real-time requirements. For this reason, if a more selective process can identify sensitive parts among the OS data structures, the impact of fault-tolerant techniques might be carefully reduced, allowing to meet the OS size and performance requirements of a real-time system.

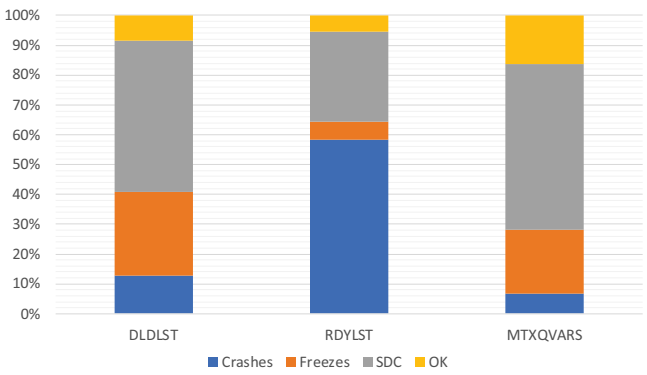

Fig. 3: Classification summary across the different target location groups

V. Fault Tolerant Online Scheduling to Improve the Resiliency of Multiprocessor-based Systems

In this paper, we assume that the system is composed of a set of tasks, and we consider that the redundancy is carried out at the task level, which means that task copies are replicated. When two identical copies of the same task are used, this approach is called *duplication*. If there is no other way to detect a fault, duplication allows a system to detect a discrepancy in results but not to decide which result is correct. If there are three task copies, we call it *triple modular redundancy* (TMR).

A. Primary/Backup Approach

1) Principle: The primary/backup (PB) approach is a method of fault-tolerant scheduling on multiprocessor embedded systems making use of two task copies: the *primary copy* (PC) and *backup copy* (BC) [27]. It is a commonly used technique for designing fault-tolerant systems owing to its easy application and minimal system overheads. Consequently, it can be used in online scheduling.

Several additional enhancements [27]–[30] to this approach have been already presented, such as the *BC deallocation* and the *BC overloading*. While the former technique frees the slot initially occupied by a backup copy when the corresponding primary copy is correctly executed, i.e., reduces system load overheads at execution time, the latter technique authorizes several backup copies to be overloaded if their respective primary copies are not scheduled on the same processor.

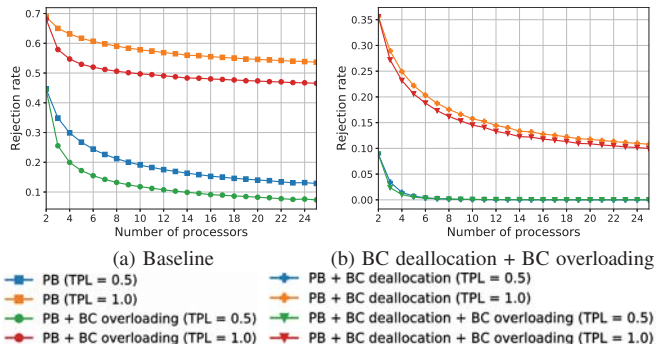

Fig. 4: Rejection rate as a function of the number of processors and targeted processor load (TPL) without fault injection

Figure 4 depicts the rejection rate, i.e., the ratio of rejected tasks to all arriving tasks to the system, as a function of the number of processors and targeted processor load (TPL). The task set at input was generated so that the targeted processor load remains the same no matter the number of processors. The higher the number of processors, the lower the rejection rate because there are more possibilities to schedule task copies. The BC overloading helps to reduce the rejection rate by several percents, but the contribution of the BC deallocation is more noteworthy: the gain is about 75% compared to the baseline PB approach and no matter whether the BC overloading is implemented or not.

2) Enhancements: In general, an operating system should promptly respond. Therefore, a choice of fault-tolerant algorithm to be implemented is important. The primary/backup approach is simple, quick, and already suitable to be used in the operating system. Nevertheless, this method can be further improved in order to be even quicker. In this section, we present our two proposed enhancements aiming at reducing the algorithm run-time [17]:

- *Limitation on the number of comparisons*: When scheduling a task, the simplest idea to cut down the algorithm run-time is to limit the number of comparisons between the free slot duration and the computation time c_i [28]. This number is computed for every task until it is definitely accepted or rejected. Every arriving task is assigned a maximum number of comparisons to search for its PC and BC slots. If this threshold is exceeded, the task is rejected. Otherwise, it is normally scheduled, i.e., accepted or rejected according to the baseline algorithm.
- *Restricted scheduling windows*: The aim of this method is threefold: (i) to avoid the mutual scheduling interference between primary and backup copies of the same task, (ii) to reduce the run-time (measured by means of the number of comparisons carried out before definitely accepting or rejecting a task), and (iii) to place the primary copies as soon as possible and the backup ones as late as possible, which increases the schedulability if the BC deallocation is enabled. A *scheduling window* for both the primary or the backup copy is a time interval (subinterval of the task window) within which the respective copy can be scheduled. The size of the scheduling window is defined by a parameter f representing the *fraction of task window*. The primary copy window of task t_i is thereby delimited by a_i and $a_i + f \cdot tw_i$ and the backup copy one by $d_i - f \cdot tw_i$ and d_i. In our algorithm, the fraction is within $0 < f \leqslant 1$, whereas it equals 1 in the conventional algorithm. An example of restricted scheduling windows with $f = 1/3$ is depicted in Figure 5.

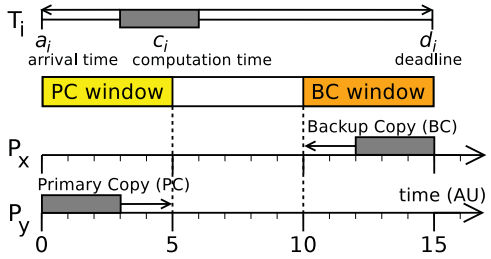

Fig. 5: PB approach with restricted scheduling windows ($f = 1/3$)

Figure 6 shows the improvement (of two aforementioned enhancements with the best parameters) in the rejection rate and in the maximum and mean numbers of comparisons per task for the PB approach with BC deallocation only. Similar results were also obtained for the PB approach with BC deallocation and BC overloading. All the proposed methods

diminish the number of comparisons per task, and most of them also decrease the rejection rate. The limitation on the number of comparisons (PC: $P/2$ comparisons; BC: 5 comparisons) reduces the algorithm run-time by 34% (mean value) and 62% (maximum value) and increases the rejection rate by only 1% compared to the primary/backup approach without any enhancing method.

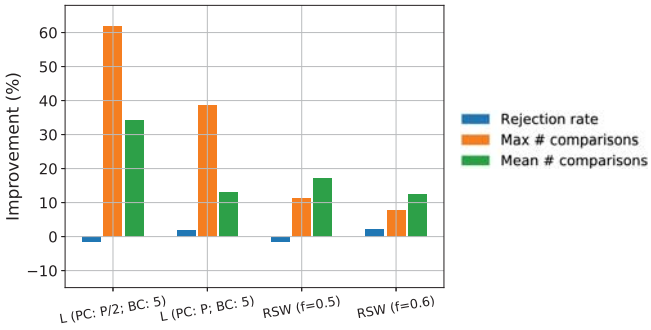

Fig. 6: Improvements to a 14-processor system compared to the baseline PB approach ($TPL = 1.0$)

3) Fault Injection: To assess the fault tolerance of the system, we inject faults with different fault rates. Figure 7 depicts the rejection rate as a function of the number of processors for the PB approach with BC deallocation and making use of the enhancing method L (PC: $P/2$ comparisons; BC: 5 comparisons). The results show that fault rates up to $1 \cdot 10^{-3}$ fault/ms have a minimal impact on the algorithm performances. This value is higher than the estimated fault rate in both standard ($2 \cdot 10^{-9}$ fault/ms [31]) and severe ($1 \cdot 10^{-5}$ fault/ms [32]) conditions. Our algorithm can therefore perform well in a harsh environment.

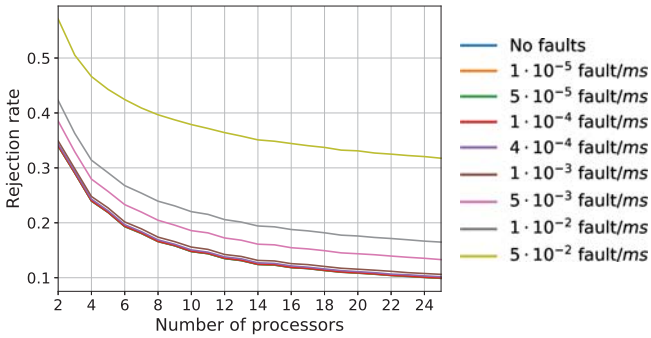

Fig. 7: Rejection rate at different fault injection rates (PB approach + BC deallocation + L (PC: $P/2$ comparisons; BC: 5 comparisons))

B. Reducing Task Replication Overheads

In general, fault detection techniques can be implemented in software or in hardware. The software ones are for example task replication, [27], [33], [34], task migration [35], [36], checkpointing [37], [38] or watchdogs [39]. As for the hardware techniques, the main idea is based on the principle to make several copies of a component [16], [19]. Regardless of the implementation level, the task replication causes the system overheads, which we will try to reduce in this section.

Therefore, we distinguish two task types: simple and double tasks. A *simple task* has one PC and one BC and a *double task* has two PCs and one BC. This differentiation avoids duplication (as is the case for the PB approach) of all tasks to detect a fault and consequently reduce the system load. Actually, the previous section showed that some tasks do not need a duplication to detect a fault because a fault is detected by timeout, no received acknowledgment, or failure of data checks. Therefore, the duplication to detect a fault is needed only for double tasks.

The principle of an online algorithm making use of two task types is summarised in Algorithm 1. The algorithm orders task using the "earliest deadline first" policy and tries to minimize the rejection rate.

Algorithm 1 Online algorithm considering simple and double tasks

Input: Mapping and scheduling of already scheduled tasks, (task t_i)
Output: Updated mapping and scheduling

```
if there is a scheduling trigger at time t then
    if a processor becomes idle and there is neither task arrival nor fault occurrence then
        if an already scheduled task copy starts at time t then
            Commit this task copy
        else
            Nothing to do
    else ▷ processor is idle and task arrives and/or fault occurs
        if a (simple or double) task t_i arrives then
            Add one or two PC_i to the task queue
        if a fault occurs during the task t_k then
            Add BC_k to the task queue
        Remove task copies having not yet started their execution
        Order the task queue
        for each task in the task queue do
            Map and schedule its task copies (PC(s) or BC)
        if an already scheduled task copy starts at time t then
            Commit this task copy
        else
            Nothing to do
```

We compare the system performance of the proposed solution with the one of a system without any fault tolerance and the one of a system using the triple modular redundancy (TMR) implemented in software. The system based on the TMR always schedules three identical task copies for each task between the task arrival time and task deadline, and the no-fault tolerant system considers only one task copy for each task. No backup copies are considered for these two systems. Our proposed solution distinguishes simple and double tasks depending on fault detection and schedules backup copies only if a fault occurs.

Figure 8 depicts the processor load as a function of the number of processors for three aforementioned systems when no fault occurs. The task set at the input is based on real data from APSS CubeSat [40]. It consists of 96% simple tasks (with a uniform distribution between 100 ms; 500 ms

for the execution time) and 4% double tasks (with a uniform distribution between $1\,ms$; $10\,ms$ for the execution time) and were all the time the same (no matter the fault tolerance level of the system). This is the reason why the processor load rate decreases (or remains constant) with the increasing number of processors. It can be seen that if a system has more than five processors, our proposed solution has similar values of the processor load as the system without any redundancy. Otherwise, the processor load of our system is lower than the one of the system without any redundancy due to some rejected tasks. The system using the TMR has a higher processor load (three times if all tasks are accepted) than the other systems because all tasks always have three task copies. The values for systems with less than 11 processors are constant because the system is fully loaded and cannot accept more tasks, which causes the task rejection. To conclude, for applications mainly consisting of simple tasks that do not need a duplication to detect a fault, since the proposed method has a similar processor load as a system which is not fault tolerant, its overheads are negligible. As it provides the operating system with fault tolerance, we recommend its implementation.

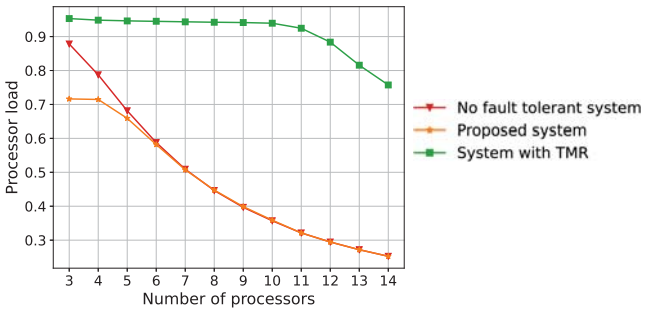

Fig. 8: Processor load for three systems with different level of fault tolerance as a function of the number of processors and without fault injection

In Figure 8, no fault is injected. However, the further results (figures not presented in this paper) show that fault rates up to $1 \cdot 10^{-4}$ fault/ms, which is higher than the worst estimated fault rate in a harsh environment (10^{-5} fault/ms [32]), have minimal impact on performances.

VI. Conclusion

In this paper, we showed the critical interplay between the reliability of a full system and the operating system's presence, either real-time or desktop. On the one hand, the experimental results highlight how significant the operating system's impact is in exposing the system to dependability issues. On the other hand, a more specific investigation pointed out where those criticalities lay when analyzing a real-time operating system. Eventually, a fault-tolerant approach applied to the operating system's scheduling task demonstrated the feasibility of targeting the operating system's fault-tolerance selectively.

Acknowledgment

This study has been achieved thanks to the financial support of the projects "IDEX Lyon OdeLe"

References

[1] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *2015 IEEE International Conference on Computer Vision (ICCV)*, Dec 2015, pp. 2722–2730.

[2] A. Vallero, A. Savino, A. Chatzidimitriou, M. Kaliorakis, M. Kooli, M. Riera, M. Anglada, G. Di Natale, A. Bosio, R. Canal, A. Gonzalez, D. Gizopoulos, R. Mariani, and S. Di Carlo, "SyRA: Early System Reliability Analysis for Cross-Layer Soft Errors Resilience in Memory Arrays of Microprocessor Systems," *IEEE Transactions on Computers*, vol. 68, no. 5, pp. 765–783, May 2019.

[3] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE analysis reliability estimates using fault-injection," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 460–469.

[4] G. S. Rodrigues, F. L. Kastensmidt, R. Reis, F. Rosa, and L. Ost, "Analyzing the impact of using pthreads versus openmp under fault injection in arm cortex-a9 dual-core," in *2016 16th European Conference on Radiation and Its Effects on Components and Systems (RADECS)*, 2016, pp. 1–6.

[5] S. Duzellier, "Radiation effects on electronic devices in space," *Aerospace Science and Technology*, vol. 9, no. 1, pp. 93 – 99, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1270963804001129

[6] Jedec. [Online]. Available: https://www.jedec.org

[7] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, no. 3, pp. 258–266, May 2005.

[8] S. Di Carlo, A. Vallero, D. Gizopoulos, G. Di Natale, A. Gonzalez, R. Canal, R. Mariani, M. Pipponzi, A. Grasset, P. Bonnot, F. Reichenbach, G. Rafiq, and T. Loekstad, "Cross-layer early reliability evaluation: Challenges and promises," in *2014 IEEE 20th International On-Line Testing Symposium (IOLTS)*, July 2014, pp. 228–233.

[9] A. Vallero, S. Tselonis, N. Foutris, M. Kaliorakis, M. Kooli, A. Savino, G. Politano, A. Bosio, G. Di Natale, D. Gizopoulos *et al.*, "Cross-layer reliability evaluation, moving from the hardware architecture to the system level: A clereco eu project overview," *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 1204–1214, 2015.

[10] M. Kooli, G. D. Natale, and A. Bosio, "Memory-aware design space exploration for reliability evaluation in computing systems," *Journal of Electronic Testing*, 03 2019.

[11] A. Savino, A. Vallero, and S. Di Carlo, "ReDO: Cross-Layer Multi-Objective Design-Exploration Framework for Efficient Soft Error Resilient Systems," *IEEE Transactions on Computers*, vol. 67, no. 10, pp. 1462–1477, Oct 2018.

[12] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, Feb. 1995.

[13] J. Carreira, H. Madeira, and J. G. Silva, "Xception: a technique for the experimental evaluation of dependability in modern computers," *IEEE Transactions on Software Engineering*, vol. 24, no. 2, pp. 125–136, Feb 1998.

[14] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, "A fast, flexible, and easy-to-develop fpga-based fault injection technique," *Microelectronics Reliability*, vol. 54, no. 5, pp. 1000–1008, 2014.

[15] G. Di Natale, D. Gizopoulos, S. Di Carlo, A. Bosio, and R. Canal, Eds., *Cross-layer reliability of computing systems*, ser. Materials, Circuits & Devices. Institution of Engineering and Technology, 2020, oCLC: 1191709900. [Online]. Available: http://public.eblib.com/choice/PublicFullRecord.aspx?p=6341977

[16] E. Dubrova, *Fault-Tolerant Design*. Springer, 2013, https://doi.org/10.1007/978-1-4614-2113-9.

[17] P. Dobiáš, "Online Fault Tolerant Task Scheduling for Real-Time Multiprocessor Embedded Systems," Ph.D. dissertation, Université de Rennes 1, 2020, https://hal.archives-ouvertes.fr/tel-03016351.

[18] P. Dobiáš, "Bibliographic Study: Mapping and Scheduling of Applications/Tasks onto Heterogeneous Faulty Processors," ENSSAT Lannion & Master of Research at ISTIC Rennes, Univ Rennes, IRISA, France, School year 2016/2017.

[19] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, Elsevier, 2007, https://doi.org/10.1016/B978-0-12-088525-1.X5000-7.

[20] (2021) Ovpsim. [Online]. Available: http://www.ovpworld.org

[21] F. Rosa, F. Kastensmidt, R. Reis, and L. Ost, "A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*, 2015, pp. 211–214.
[22] Q. Xu, T. Mytkowicz, and N. S. Kim, "Approximate computing: A survey," *IEEE Design Test*, vol. 33, no. 1, pp. 8–22, 2016.
[23] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August, "Design and evaluation of hybrid fault-detection systems," in *32nd International Symposium on Computer Architecture (ISCA'05)*, 2005, pp. 148–159.
[24] D. Mamone, A. Bosio, A. Savino, S. Hamdioui, and M. Rebaudengo, "On the analysis of real-time operating system reliability in embedded systems," in *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, 2020, pp. 1–6.
[25] Freertos. [Online]. Available: https://www.freertos.org/index.html
[26] EEMBC, *Autobench - Software benchmark data book*. www.eembc.org: EEMBC, 2015.
[27] S. Ghosh, R. Melhem, and D. Mosse, "Fault-Tolerance Through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 3, March 1997, pp. 272–284, https://doi.org/10.1109/71.584093.
[28] M. Naedele, "Fault-Tolerant Real-Time Scheduling under Execution Time Constraints," in *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA)*, 1999, pp. 392–395, https://doi.org/10.1109/RTCSA.1999.811286.
[29] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "A New Fault-Tolerant Scheduling Technique for Real-Time Multiprocessor Systems," in *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*, 1995, pp. 197–202, https://doi.org/10.1109/RTCSA.1995.528772.
[30] Q. Zheng, B. Veeravalli, and C.-K. Tham, "On the Design of Fault-Tolerant Scheduling Strategies Using Primary-Backup Approach for Computational Grids with Low Replication Costs," in *IEEE Transactions on Computers*, vol. 58, no. 3, 2009, pp. 380–393, https://doi.org/10.1109/TC.2008.172.
[31] S. Du, E. Zio, and R. Kang, "A New Analytical Approach for Interval Availability Analysis of Markov Repairable Systems," in *IEEE Transactions on Reliability*, vol. 67, no. 1, March 2018, pp. 118–128, https://doi.org/10.1109/TR.2017.2765352.
[32] R. M. Pathan, "Real-Time Scheduling Algorithm for Safety-Critical Systems on Faulty Multicore Environments," in *Real-Time Systems*, vol. 53, no. 1, 2017, pp. 45–81, https://doi.org/10.1007/s11241-016-9258-z.
[33] S. Wang, K. Li, J. Mei, G. Xiao, and K. Li, "A Reliability-aware Task Scheduling Algorithm Based on Replication on Heterogeneous Computing Systems," in *Journal of Grid Computing*, vol. 15, no. 1, 03 2017, pp. 23–39, https://doi.org/10.1007/s10723-016-9386-7.
[34] M. A. Haque, H. Aydin, and D. Zhu, "On Reliability Management of Energy-Aware Real-Time Systems Through Task Replication," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, March 2017, pp. 813–825, https://doi.org/10.1109/TPDS.2016.2600595.
[35] J. Mei, K. Li, X. Zhou, and K. Li, "Fault-Tolerant Dynamic Rescheduling for Heterogeneous Computing Systems," in *Journal of Grid Computing*, vol. 13, no. 4, 2015, pp. 507–525, https://doi.org/10.1007/s10723-015-9331-1.
[36] M. Hasan and M. S. Goraya, "A Framework for Priority Based Task Execution in the Distributed Computing Environment," in *International Conference on Signal Processing, Computing and Control (ISPCC)*, 2015, pp. 155–158, https://doi.org/10.1109/ISPCC.2015.7375016.
[37] M. Fayyaz and T. Vladimirova, "Fault-Tolerant Distributed approach to satellite On-Board Computer design," in *2014 IEEE Aerospace Conference*, March 2014, pp. 1–12, https://doi.org/10.1109/AERO.2014.6836199.
[38] M. Singh, "Performance Analysis of Checkpoint Based Efficient Failure-Aware Scheduling Algorithm," in *International Conference on Computing, Communication and Automation (ICCCA)*, 2017, pp. 859–863, https://doi.org/10.1109/CCAA.2017.8229916.
[39] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Springer, 2006, https://doi.org/10.1007/0-387-32937-4.
[40] P. Dobiáš, E. Casseau, and O. Sinnen, "Evaluation of Fault Tolerant Online Scheduling Algorithms for CubeSats," in *23rd Euromicro Conference on Digital System Design (DSD'20)*, August 2020, https://doi.org/10.1109/DSD51259.2020.00102.