Improving the CubeSat Reliability Thanks to a Multiprocessor System using Fault Tolerant Online Scheduling

Petr Dobiáš, Emmanuel Casseau, Oliver Sinnen

Improving the CubeSat Reliability Thanks to a Multiprocessor System using Fault Tolerant Online Scheduling

Petr Dobiáš and Emmanuel Casseau
Univ Rennes, Inria, CNRS, IRISA, Lannion, France
Oliver Sinnen
Department of Electrical, Computer and Software Engineering, University of Auckland
Auckland, New Zealand

https://doi.org/10.1016/j.micpro.2021.104312

Abstract

More than 21 000 objects fly in outer space and are exposed to the harsh space environment. The size of space objects considerably varies. Our research focuses on small satellites, such as CubeSats, which have to respect time, spatial and energy constraints. To tackle this issue, this paper presents and evaluates two fault tolerant online scheduling algorithms: the algorithm scheduling all tasks as aperiodic (called OneOff) and the algorithm placing arriving tasks as aperiodic or periodic tasks (called OneOff&Cyclic). Based on several scenarios, the results show that the performance of ordering policies are influenced by the system load and the proportions of simple and double tasks to all tasks to be executed. The "Earliest Deadline" and "Earliest Arrival Time" ordering policies for OneOff or the "Minimum Slack" ordering policy for OneOff&Cyclic reject the least tasks in all tested scenarios. The paper also deals with the analysis of scheduling time to evaluate real-time performance of ordering policies and shows that OneOff requires less time to find a new schedule than OneOff&Cyclic. Finally, it was found that the studied algorithms perform well also in a harsh environment and provide the same reliability level as systems based on triple modular redundancy with very much less system power consumption.

1 Introduction

The website https://www.n2yo.com/ tracks objects that fly in outer space. As of October 28, 2020, its database counts 21676 objects. The size of space objects ranges from the International Space Station (ISS), through the Hubble Space Telescope to very small satellites. Such very small satellites can be classified

according to their weights into different categories. One possible classification distinguishes: minisatellite (100 kg to 180 kg), microsatellite (10 kg to 100 kg), nanosatellite (1 kg to 10 kg), picosatellite (0.01 kg to 1 kg) and femtosatellite (0.001 kg to 0.01 kg) [23].

In order to visualise the difference in weight and size, Figure 1 depicts the mass of a satellite as a function of its volume for several satellites. In this figure, we also plot three ellipses encompassing different implementations of fault tolerance.

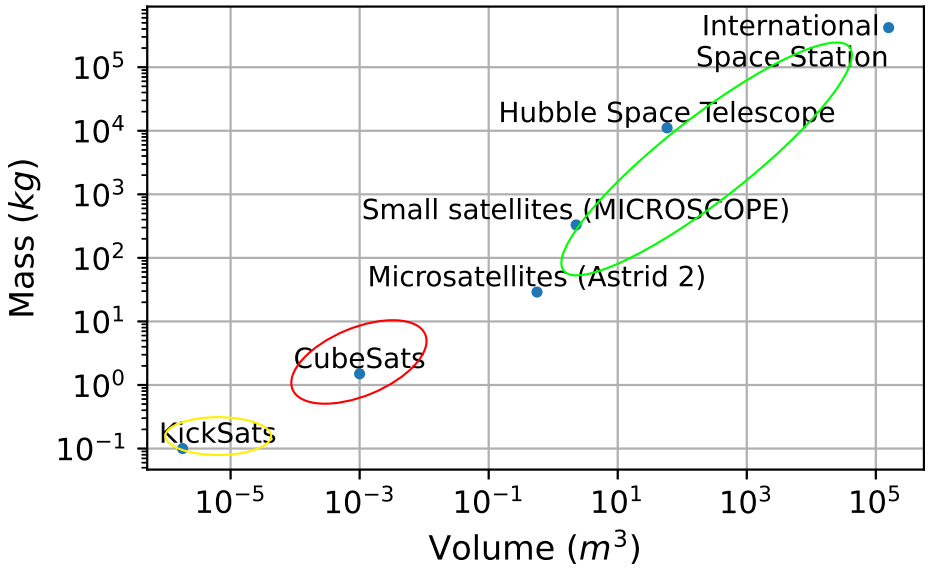


Figure 1: Comparison of satellites

The satellites situated within the **green ellipse** have no significant constraints on space and weight. Consequently, the fault tolerance can be put into practice by using hardware redundancy in space, i.e. the components are for example triplicated, their outputs are compared and the majority result is chosen, which is the principle of the well-known triple modular redundancy (TMR).

The **yellow ellipse** incorporates tiny satellites, such as KickSats or ChipSats. These satellites are printed circuit boards having several square centimetres. Due to the restricted size and limited energy harvesting, hardware space redundancy is not feasible. If the fault tolerance is considered at all, it can be thereby implemented in software.

The **red ellipse** includes the satellites that are bigger and heavier than KickSats but smaller and lighter than microsatellites. A typical example of this category is a CubeSat, which will be described in the next section. These satellites still have space and weight constraints and consequently hardware space redundancy is not possible. Nevertheless, since they are bigger than KickSats, the fault tolerance can be put into practice at the software level.

Regarding this trade-off between physical aspects (weight, size and energy) and fault tolerance, CubeSats are in the centre of our interest. Taking into account all constraints, such as time, reliability or energy, the mapping and scheduling of tasks or applications to be executed on such devices represent a challenging problem.

Last but not least, technology has been progressively under development and, as the author of [20] suggests, it might be better to make use of one state-of-the-art integrated commercial off-the-shelf (COTS) chip, especially for missions with limited budget. In fact, it can take advantage of redundancy thanks to its several processors and function better than one outdated single processor chip even if it was designed for space missions.

1.1 CubeSats

The idea of CubeSats dates back to 1999 and its aim was to provide affordable access to space by defining standard dimensions to reduce costs and time [22]. At present, CubeSats become more and more popular, their number of launches increases and they are built not only at universities but also by companies and space agencies [12].

CubeSats are small satellites consisting of several units (e.g. 1U, 2U, 3U or 6U) where each unit (1U) is a 10 cm cube, which can weigh up to 1.3 kg [22]. CubeSats are composed of several systems, such as on-board computer, electrical power system, attitude determination and control system, communication system, and payload. Their missions are aimed at scientific investigations, like studying urban heat islands [1] or polar auroras and airglow [7].

CubeSats operate in the harsh space environment, where they are exposed to charged particles and radiations. These phenomena cause both transient effects, such as single event upsets, and long-term effects, e.g. total ionising dose [18]. Consequently, it is necessary for CubeSats to be robust against faults to achieve their mission.

1.2 Proposed Solution

Our aim is to provide CubeSats with fault tolerance. As there are several systems aboard CubeSats and most of them has its own processor, we present a solution gathering all processors on one board. This modification will reduce space and weight and improve the system resilience. First, a shielding against radiation will be easier to put into practice [2]. Second, a CubeSat will remain operational even in case of a permanent processor failure because processors are not dedicated to one system (as it is done aboard current CubeSats) and each processor can execute any task. Although this implementation choice may seem considerable, it was successfully realised on board of ArduSat, which counts 17 processors on one board [15].

Once all processors are gathered on one board, we intend to use the proposed scheduling algorithms dealing with all tasks (no matter the system) on board of any CubeSat or any small satellite. These algorithms schedule all types of tasks

(periodic, sporadic and aperiodic), detect faults and take appropriate measures to provide correct results. They are executed online in order to promptly manage occurring faults and respect real-time constraints. They are mainly meant for CubeSats based on commercial-off-the-shelf processors, which are not necessarily designed to be used in space applications and therefore more vulnerable to faults than radiation hardened processors [26].

1.3 Our Contributions and Paper Organisation

The contributions of this paper are as follows:

- we analyse the data from two real CubeSats in terms of the workload;
- we assess the algorithm performance using the rejection rate (which represents the ratio of rejected tasks to all arriving tasks and which we try to minimise) for different scenarios; whenever possible the results are compared to the optimal solution provided by CPLEX solver;
- we analyse the scheduling time of ordering policies for two studied algorithms;
- we evaluate how the algorithms deal with faults;
- we assess the use of buffer to reduce the number of scheduling searches;
- 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;
- based on the algorithm performance, we suggest which algorithm should be used on board of the CubeSat.

The remainder of this paper is organised as follows. Section 2 sums up the related work on fault tolerance in CubeSats and Section 3 presents our system, task and fault models. The algorithms are described in Section 4. Section 5 then introduces the experimental framework and the results are analysed in Section 6. Section 7 concludes this paper.

2 Fault Tolerance in CubeSats

This section summarises how the fault tolerance is put into practice on board of CubeSats.

First of all, we stress that not all CubeSats are fault tolerant mainly due to financial or time constraints [14]. If a CubeSat is made more robust, the fault tolerance is implemented rather in hardware than in software. Though, a usual hardware technique is redundancy of several or all components [19], 43% of CubeSats do not use any redundancy due to budget, time or space constraints [13]. Actually, the triple modular redundancy (TMR) is a standard aboard

aircraft or bigger satellites [11] but it has a significant system overheads [26] that makes it unsuitable for small satellites such as CubeSats.

Other fault tolerant techniques aboard CubeSats are for example watchdog timers [28] or data protection techniques [6]. It is also possible to analyse error reports, scan important parameters, like power consumption or temperature, in order to detect abnormal behaviour and send appropriate commands if necessary [19, 6, 5].

Cerrolaza et al. presented a survey on multi-core devices for safety-critical systems [4]. This thorough survey considers several device levels (nanoscale, component and device) and it also overviews avionics and space domains.

Even though software techniques are not common in CubeSats, they are widely used in other applications, e.g. creation of several task copies [16, 27] or task rescheduling [21] or regular checkpointing [25].

3 System, Fault and Task Models

Similarly to the model in [9], the studied system consists of P interconnected identical processors[1]. It handles all tasks on board of the CubeSat. These tasks are mostly related to housekeeping (like sensor measurements), communication with ground station and storing or reading data from memory.

The task model distinguishes aperiodic and periodic tasks. An *aperiodic task*, depicted in Figure 2, is characterised by arrival time a_i, execution time et_i, deadline d_i and task type tt_i, which will be defined in the next paragraph. A *periodic task*, represented in Figure 3, has several instances and has four attributes: ϕ_i (which is the arrival time of the first instance), execution time et_i, period T_i and task type tt_i. We consider that the relative deadline equals the period. For both aperiodic and periodic tasks, a task must be executed before deadline or beginning of the next period, respectively.

As for the fault model, it considers both transient and permanent faults and it distinguishes two task types: simple (S) and double (D) tasks depending on the fault detection. For both task types, we differentiate two types of task copies: *primary copy* (PC) and *backup copy* (BC). The former copies are necessary for task execution in a fault-free environment. If a primary copy is faulty, the corresponding backup copy is scheduled. *Simple tasks* have only one PC because a fault is detected by timeout, no received acknowledgment or failure of data checks. By contrast, the fault detection for *double tasks* requires the execution of two PCs[2] and then their comparison because fault detection techniques for simple tasks may not be sufficient to detect a fault. We consider that a scheduler is robust, e.g. data related to scheduling, such as task queues, are duplicated in memory or the system has a spare one if necessary.

[1] To simplify, a system presented in this paper is composed of homogeneous processors sharing the same memory. Nevertheless, this model can be easily extended to a system with heterogeneous processors, like in [29].

[2] Two task copies of the same task t_i can overlap each other on different processors but it is not necessary. However, they must not be executed on one processor in order to be able to detect a faulty processor.

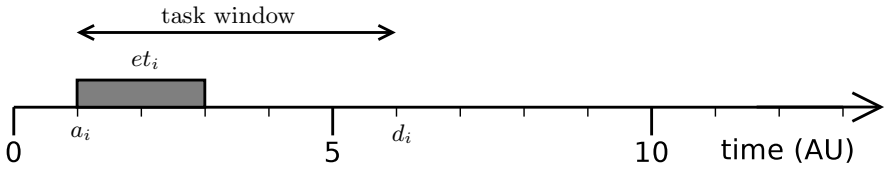


Figure 2: Model of aperiodic task t_i

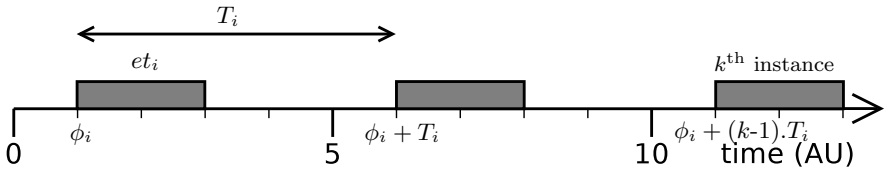


Figure 3: Model of periodic task τ_i [3]

Our objective is to minimise the task rejection rate subject to real-time and reliability constraints, which means maximising the number of tasks being correctly executed before deadline even if a fault occurs.

4 Presentation of Algorithms

This section describes two algorithms meant for global scheduling on multiprocessor systems. First of all, it starts with several general principles applicable for both of them.

All tasks arriving to the system are ordered in a task queue using different policies. The policies for aperiodic tasks are as follows: Random, Minimum Slack (MS) first, Highest ratio of et_i to (d_i-t) first, Lowest ratio of et_i to (d_i-t) first, Longest Execution Time (LET) first, Shortest Execution Time (SET) first, Earliest Arrival Time (EAT) first and Earliest Deadline (ED) first; and the ones for periodic tasks are as reads: Random, Minimum Slack (MS) first, Longest Execution Time (LET) first, Shortest Execution Time (SET) first, Earliest Phase (EP) first and Rate Monotonic (RM).

A preemption is not authorised but the task rejection is allowed. A task t_i is rejected at time t and removed from the task queue if its task copies do not meet its deadline, i.e. $t + et_i > d_i$ for the aperiodic task or $t + et_i > \phi_i + k \cdot T_i$ for the k^{th} instance of periodic task. We remind the reader that a simple task t_i has one PC (denoted by PC_i), whereas a double task t_i has two PCs (labeled respectively $PC_{i,1}$ and $PC_{i,2}$) in a fault-free environment.

As Figure 4 shows, all primary copies are scheduled as soon as possible to avoid idle processors just after the task arrival and possible high processor load later. As our goal is to minimise the task rejection, the algorithm reserves a certain time of the task window to place a backup copy if the PC execution is faulty. The end of the PC scheduling window is defined as $d_i - \alpha \cdot et_i$ for the aperiodic task and $\phi_i + k \cdot T_i - \alpha \cdot et_i$ for the k^{th} instance of periodic task (with

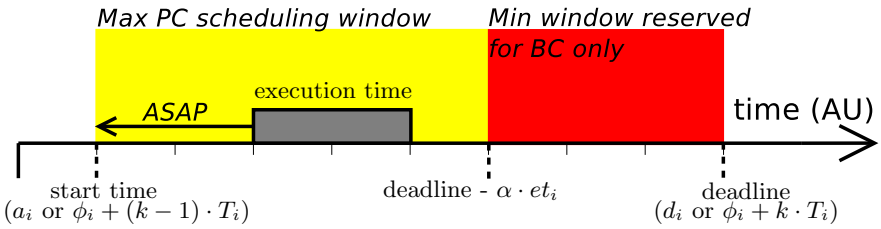


Figure 4: Principle of scheduling task copies

$\alpha \geqslant 1$). In this paper, we consider without lost of generality that $\alpha = 1$. If a primary copy is found out faulty, the corresponding backup copy is scheduled and can start its execution immediately, i.e. even during the PC scheduling window, because its results are necessary.

As for the processor allocation, we call a *slot*, a time interval on processor schedule. The algorithm starts to check the first free slot on each processor and then, if a solution was not found, it continues with next slots (second, third, ...) until a solution is obtained or all free slots on all processors are tested. The principle of the search is illustrated in Figure 5, where xC_i stands for primary or backup copy of task t_i. We chose this processor allocation policy because the analysis in [8] showed that this policy is the best when scheduling a task copy on a multiprocessor system.

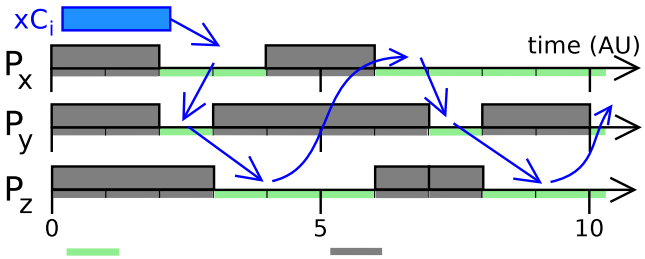


Figure 5: Principle of algorithm search for a free slot

4.1 Mathematical Programming Formulation

We define the mathematical programming formulation of the studied scheduling problem as follows:

$$\max \sum_{i}^{\text{Set of tasks}} t_i \text{ is accepted}$$

subject to

$$\begin{cases} \textbf{1)}\ a_i \leqslant start(PC_i) < end(PC_i) \leqslant d_i - et_i \\ \textbf{2)}\ \begin{cases} \text{For simple tasks: } PC_i \in P_x \Rightarrow BC_i \notin P_x \\ \text{For double tasks: } PC_{i,1} \in P_x \Rightarrow (PC_{i,2} \notin P_x \text{ and} \\ \qquad\qquad BC_i \notin P_x) \text{ and } PC_{i,2} \in P_y \Rightarrow BC_i \notin P_y \end{cases} \\ \textbf{3)}\ (xC_i \text{ and } xC_j) \in P_x \Rightarrow \\ \quad end(xC_i) \leqslant start(xC_j) \text{ or } end(xC_j) \leqslant start(xC_i) \\ \textbf{4)}\ \text{For double tasks: } PC_{i,1} \text{ scheduled } \Leftrightarrow PC_{i,2} \text{ scheduled} \end{cases}$$

The objective function is to maximise the number of accepted tasks, which is equivalent to minimise the task rejection rate. The first constraint is related to the PC scheduling window depicted in Figure 4 and the second one forbids task copies of the same task to be scheduled on the same processor. The third constraint stands for no overlap among task copies xC (i.e. PC or BC) on one processor, i.e. only one task copy can be scheduled per processor at the same time. The last constraint requires that both primary copies of double tasks are scheduled.

4.2 Online Scheduling Algorithm for All Tasks Scheduled as Aperiodic Tasks (OneOff)

The online algorithm scheduling arriving tasks as aperiodic ones is called OneOff in this paper. When it is used, all tasks are considered as aperiodic, which means that each instance of periodic task is transformed into an aperiodic task[3].

The principle of OneOff is summarised in Algorithm 1.

First (Line 1), the algorithm is triggered if (i) a processor becomes idle, (ii) a processor is idle and a task arrives, or (iii) a fault occurs.

If there is neither task arrival nor fault occurrence and a processor becomes/is idle (i.e. Case (i)), a new search for a schedule is not necessary and task copies are committed using an already defined schedule (Lines 2-6).

Otherwise (Lines 7-19), new task copies (PC(s) for new task and BC for task impacted by fault) are added to the task queue. Then, the algorithm removes all task copies that have not yet started their execution, it orders tasks in the queue using the chosen ordering policy and it searches for a new schedule. Finally (Lines 16-19), the task copies starting at time t are committed.

The complexity for one search for a schedule where N is the number of tasks in the task queue and P is the number of processors is as follows. The

[3]The arrival time a_i equals $\phi_i + (k-1) \cdot T_i$ and the deadline d_i is computed as $a_i + T_i$. The execution time et_i and the task type tt_i are not modified.

Algorithm 1 Principle of online algorithm scheduling all tasks as aperiodic tasks (OneOff)

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

```
1: if there is a scheduling trigger at time t then
2:     if a processor becomes idle and there is neither task arrival nor fault occurrence then
3:         if an already scheduled task copy starts at time t then
4:             Commit this task copy
5:         else
6:             Nothing to do
7:     else                          ▷ processor is idle and task arrives and/or fault occurs
8:         if a (simple or double) task t_i arrives then
9:             Add one or two PC_i to the task queue
10:        if a fault occurs during the task t_k then
11:            Add BC_k to the task queue
12:        Remove task copies having not yet started their execution
13:        Order the task queue
14:        for each task in the task queue do
15:            Map and schedule its task copies (PC(s) or BC)
16:        if an already scheduled task copy starts at time t then
17:            Commit this task copy
18:        else
19:            Nothing to do
```

complexity to order a task queue is $O(N \log(N))$ and the one to add a task in an already ordered queue is $O(N)$. Then, it takes $O(P \cdot N \cdot (\#\ task\ copies))$ to map and schedule tasks from the task queue and $O(1)$ to commit a task copy. If we consider that the task queue is always ordered, the overall worst-case complexity is as follows:

$$O(N + P \cdot N \cdot (\#\ task\ copies) + 1) \tag{1}$$

4.3 Method to Reduce the Number of Scheduling Searches

If there is at least one processor available, OneOff carries out a new search for a schedule at every task arrival, which may cause rather high number of scheduling searches. The maximum theoretical number of scheduling searches can be computed as the sum of the number of tasks at the system input and the number of task copies.

In order to reduce this number, we present a method making use of a buffer, which is a commonly used technique in scheduling [10, 17]. It computes the slack for every task t_i and checks whether or not a search for a new schedule can be postponed. The *slack* stands for the remaining time between the current time and the task deadline. The slack is called *short* if

$$d_i - \text{current time} - et_i \leqslant K \cdot et_i \qquad \text{where } K \in \mathbb{N} \tag{2}$$

otherwise, it is called *large*.

The principle of the method is illustrated in Figure 6. The highlighted background shows the part that is added to the baseline version. To enter it, the algorithm checks the slack using Formula 2 where $K = \beta$ and the current time equals the task arrival time. If the computed slack is large, the task is put into the buffer. Otherwise, it is scheduled as usual.

The tasks stored in the buffer of length L are scheduled if the buffer is full. In order to regularly check slacks of tasks queuing in the buffer, a verification (with $K = \gamma$) is carried out if a new task arrives in the buffer or a processor becomes idle. If any task has a short slack, the buffer is emptied and all tasks are scheduled.

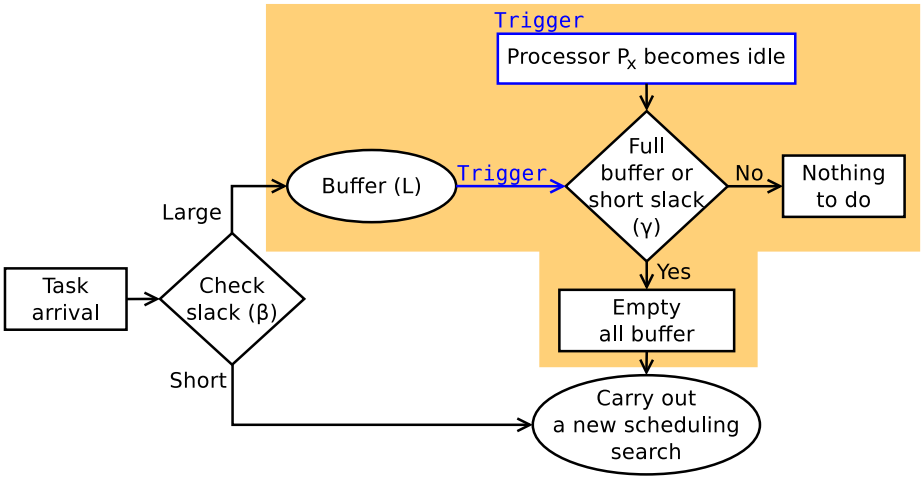


Figure 6: Principle of the method to reduce the number of scheduling searches

4.4 Online Scheduling Algorithm for All Tasks Scheduled as Aperiodic or Periodic Tasks (OneOff&Cyclic)

The online algorithm scheduling arriving tasks as aperiodic or periodic tasks is called OneOff&Cyclic. It is aware that there are not only aperiodic tasks but also periodic ones. Therefore, there are two task sets: one for periodic tasks and one for aperiodic ones.

The principle of OneOff&Cyclic is summed up in Algorithm 2.

First (Line 1), the algorithm is triggered (i) if a processor becomes idle, and/or if there is (ii) an arrival of aperiodic task(s), (iii) an arrival/withdrawal[4] of periodic task(s), or (iv) a fault during task execution.

[4]A possibility to add and withdraw a periodic task from the task set allows us to model sporadic tasks related to the communication between a CubeSat and a ground station. More details are presented in Section 5.

Algorithm 2 Principle of online algorithm scheduling all tasks as periodic or aperiodic tasks (OneOff&Cyclic)

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

```
1: if there is a scheduling trigger at time t then
2:     if a processor becomes idle and there is neither
       arrival/withdrawal of periodic task nor arrival of aperiodic task nor fault
   occurrence then
3:         if an already scheduled task copy starts at time t then
4:             Commit this task copy
5:         else
6:             Nothing to do
7:     else▷ processor is idle and there is a change in set of periodic or aperiodic
   tasks and/or a fault occurs
8:         if a periodic task t_i arrives or is withdrawn then
9:             Add/withdraw one or two PC_i to/from the queue of periodic
   tasks
10:        if an aperiodic task t_i arrives then
11:            Add one or two PC_i to the queue of aperiodic tasks
12:        if a fault occurs during the task t_k then
13:            Add BC_k to the queue of aperiodic tasks
14:        Remove task copies having not yet started their execution
15:        Order the task queues
16:        for each task in the task queue of aperiodic tasks do
17:            Map and schedule its task copies (PC(s) or BC)
18:        for each task in the task queue of periodic tasks do
19:            Map and schedule its task copies (PC(s) or BC)
20:        if an already scheduled task copy starts at time t then
21:            Commit this task copy
22:        else
23:            Nothing to do
```

In the case a processor becomes/is idle (Case (i)), a new search for a schedule is not carried out and task copies are committed using an already defined schedule (Lines 2-6). As there is no modification in task sets, the schedule of one hyperperiod, which is the least common multiple of task periods, is repeated until one of Cases (ii)-(iv) happens.

Otherwise (Lines 7-23), the task sets of periodic and aperiodic tasks are updated and all task copies that have not yet started their execution are removed from the former schedule. Afterwards (Lines 15-19), tasks are ordered and the algorithm schedules aperiodic tasks and periodic ones. Finally (Lines 20-23), the task copies starting at time t are committed.

Similarly to OneOff, we denote N_{aper} as the number of aperiodic task in the task queue and N_{per} as the number of task instances per hyperperiod of

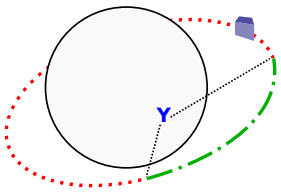
periodic tasks in the task queue. The overall worst-case complexity is as reads:

$$O(N_{aper} + P \cdot N_{aper} \cdot (\#\ task\ copies) + N_{per} + P \cdot N_{per} \cdot (\#\ task\ copies) + 1) \quad (3)$$

5 Experimental Framework

When a CubeSat orbits the Earth, two main phases can be identified from the scheduling point of view: *communication* and *no-communication phases*. During the no-communication phase (marked by red dashed line in Figure 7), there is no communication between a CubeSat and a ground station and the CubeSat mainly executes periodic tasks associated with for example telemetry, reading/storing data or checks. If there is an interrupt due to an unexpected or asynchronous event, it is considered as an aperiodic task. When a communication with a ground station is possible, i.e. during the communication phase, periodic tasks related to the communication are executed in addition to the previously mentioned tasks.

One CubeSat orbit around the Earth generally lasts for about 95 minutes and the duration of communication between a CubeSat and a ground station takes roughly 10 minutes per one orbit. Nonetheless, it may also happen that there is no communication due to CubeSat trajectory.



One orbit duration around the Earth:
about 95 min

Communication duration:
about 10 min

Figure 7: Communication phase (green dot-and-dash line) and no-communication phase (red dashed line)

The data used in our experimental framework are based on real CubeSat data provided by the Auckland Program for Space Systems (APSS)[5], based on Cortex-M3 processors, and by the Space Systems Design Lab (SSDL)[6], using Atmel AVR32 microcontrollers. These data were gathered by functionality and generalised in order to generate more data for our simulations. They are respectively called Scenario APSS and Scenario RANGE and summarised in Tables 1 and 2, where U denotes a uniform distribution and one hyperperiod is the least common multiple of task periods.

In order to further analyse the algorithm performance (see Section 6), we also modified Scenario APSS. This scenario is called Scenario APSS-modified. Its tasks are the same as for Scenario APSS but the periods of $500ms$ were prolonged

[5] `https://space.auckland.ac.nz/auckland-program-for-space-systems-apss/`
[6] `http://www.ssdl.gatech.edu/`

Table 1: Set of tasks for Scenario APSS

Periodic tasks

Function	**Task type**	**Phase** ϕ_i	**Period** T_i	**Execution time** et_i	**# of tasks**
Communication	D	U(0; T)	500 ms	U(1ms; 10ms)	2
Reading data	S	U(0; T)	1000 ms	U(100ms; 500ms)	10
Telemetry	D	U(0; T)	5000 ms	U(1ms; 10ms)	2
Storing data	S	U(0; T)	10000 ms	U(100ms; 500ms)	7
Readings	D	U(0; T)	60000 ms	U(1ms; 10ms)	2

Sporadic tasks related to communication

Function	**Task type**	**Phase** ϕ_i	**Period** T_i	**Execution time** et_i	**# of tasks**
Communication	S	U(0; T)	500 ms	U(1ms; 10ms)	46

Aperiodic tasks

Function	**Task type**	**Arrival time** a_i	**Execution time** et_i	**# of tasks**
Interrupts	D	U(0; 100000ms)	U(1ms; 10ms)	1

Table 2: Set of tasks for Scenario RANGE

Periodic tasks

Function	**Task type**	**Phase** ϕ_i	**Period** T_i	**Execution time** et_i	**# of tasks**
Kalman filter	D	U(0; T)	100 ms	U(1ms; 30ms)	1
Attitude control	D	U(0; T)	100 ms	U(10ms; 30ms)	1
Sensor polling	D	U(0; T)	100 ms	U(1ms; 5ms)	5
Telemetry gathering	S	U(0; T)	20000 ms	U(100ms; 500ms)	1
Telemetry beaconing	S	U(0; T)	30000 ms	U(10ms; 100ms)	2
Self-check	D	U(0; T)	30000 ms	U(1ms; 10ms)	5

Sporadic tasks related to communication

Function	**Task type**	**Phase** ϕ_i	**Period** T_i	**Execution time** et_i	**# of tasks**
Communication	S	U(0; T)	500 ms	U(1ms; 10ms)	10

Aperiodic tasks

Function	**Task type**	**Arrival time** a_i	**Exec. time** et_i	**# of tasks**
Interrupts, GPS	D	U(0; 10000ms)	U(1ms; 50ms)	10

to 1000 ms and periods longer than 5000 ms were shortened to 5000 ms. The number of tasks, whose periods were modified, per period were computed pro rata. Thus, the system load and the proportion of simple and double tasks for Scenarios APSS and APSS-modified are the same.

To model dynamic aspect, although task sets are defined in advance for simulations, they are unknown to the algorithms until discrete simulation time equals the arrival time (for aperiodic tasks) or phase (for periodic and sporadic tasks).

To evaluate the algorithms, 20 simulations of 2 hyperperiods were realised and the obtained values were averaged.

To compare our results, resolutions carried out in CPLEX solver[7] (described in Section 4.1) were computed based on the same data set. To model real-time aspect (i.e. dynamic task arrival) in CPLEX solver, at each task arrival, the main function updates data (arrival/withdrawal of periodic task and/or arrival of aperiodic task) and launches a new resolution using the current data set. Due to computational time constraints in this case, only results for One-Off&Cyclic were obtained and no fault was injected.

For simulations with fault injection, we take into account that the worst estimated fault rate in the real space environment is 10^{-5} fault/ms [24]. Therefore, we inject faults at the level of task copies with fault rate for each processor between $1 \cdot 10^{-5}$ and $1 \cdot 10^{-3}$ fault/ms in order to assess algorithm performance not only using the real fault rate but also its higher values. For the sake of simplicity, we consider only transient faults and that one fault can impact at most one task copy.

Regarding the metrics, we make use of the *rejection rate*, which is the ratio of rejected tasks to all arriving tasks, and the *system throughput*, which counts the number of correctly executed tasks. In a fault-free environment, this metric is equal to the number of tasks minus the number of rejected tasks. The *task queue length* stands for the number of tasks in the task queue, which are about to be ordered and scheduled. The algorithm run-time is measured by the *scheduling time*, which is the time elapsed during one scheduling search. Finally, we evaluate the *number of scheduling searches*, i.e. how many times a search for a new schedule was carried out.

6 Results

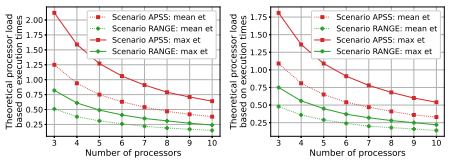
In this section, firstly, we analyse the data sets based on two real CubeSats. Secondly, we compare the rejection rate for both algorithms. Thirdly, we evaluate the buffer to reduce the number of scheduling searches. Fourthly, we analyse the scheduling time of each ordering policy for both algorithms. Fifthly, we evaluate the algorithm performance in the presence of faults. Sixthly, we compare our proposed solution with a system without any fault tolerance, as it is currently done in CubeSats, and a system using the triple modular redundancy (TMR), as it is usually implemented as a standard in bigger satellites.

[7] `https://www.ibm.com/analytics/cplex-optimizer`

6.1 Preliminary Analysis

Before analysing separately the performance of OneOff and OneOff&Cyclic, we focus on the system load and task proportions for each scenario in a fault-free environment.

Based on Tables 1 and 2, we computed the theoretical processor load when considering both maximum and mean execution times of each task. We remind the reader that a simple task has one primary copy and a double task has two primary copies. The results are depicted in Figure 8 representing the processor load respectively for both communication phases as a function of the number of processors.



(a) Communication phase (b) No-communication phase

Figure 8: Theoretical processor load when considering maximum and mean execution times (et) of each task

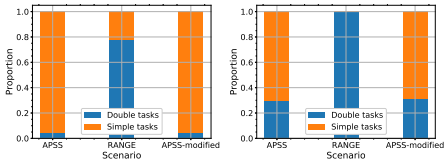
Scenario RANGE has lower theoretical processor load than other two scenarios no matter the communication phase. Theoretically, it means that all tasks for Scenario RANGE can be scheduled (maximum theoretical processor load is between 22% for 10-processor systems and 82% for 3-processor systems) while it is not always possible for Scenarios APSS and APSS-modified because the maximum theoretical processor load exceeds 100% when a CubeSat has only a few processors.

Regarding the proportion of simple and double tasks, they are represented in Figure 9. It can be observed that during the communication phase the percentage of double tasks for Scenarios APSS and APSS-modified is low (about 4%) while the task set for Scenario RANGE consists of 78% double tasks. During the no-communication phase, the percentage of simple tasks is almost negligible (0.02%) for Scenario RANGE and it is about 30% for other two scenarios.

To conclude, our experimental framework makes use of two very different sets of scenarios. On the one hand, Scenarios APSS and APSS-modified have high system load and high proportion of simple tasks compared to double tasks. On the other hand, Scenario RANGE contains mainly double tasks and has lower system load.

6.2 Rejection Rate of OneOff and OneOff&Cyclic

We compare different ordering policies for three scenarios to choose which policy is the best in terms of the rejection rate.

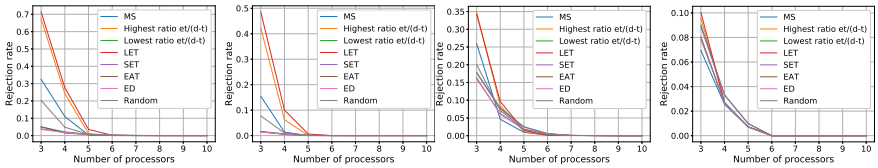


(a) Communication phase (b) No-communication phase

Figure 9: Proportions of simple and double tasks

6.2.1 Analysis of OneOff

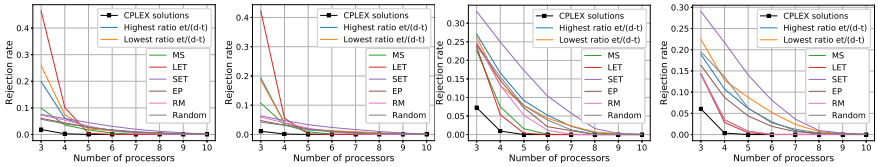
Figure 10 shows the rejection rate of Scenarios APSS and APSS-modified for both communication phases as a function of the number of processors. Scenario RANGE is not presented because the rejection rate is 0 regardless of ordering policy and communication phase. This is due to the task data set, which has rather low system load. We notice that the "Earliest Deadline" or "Earliest Arrival Time" techniques overall reject the least tasks.



(a) Scenario APSS; com. phase (b) Scenario APSS-mod.; com. ph. (c) Scenario APSS; no-com. phase (d) Scenario APSS-mod.; no-com. ph.

Figure 10: Rejection rate of OneOff as a function of the number of processors

6.2.2 Analysis of OneOff&Cyclic



(a) Scenario APSS; com. phase (b) Scenario APSS-mod.; com. ph. (c) Scenario APSS; no-com. phase (d) Scenario APSS-mod.; no-com. ph.

Figure 11: Rejection rate of OneOff&Cyclic as a function of the number of processors

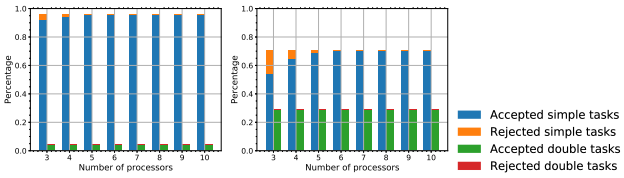
Figure 11 depicts the rejection rate of Scenarios APSS and APSS-modified

for both communication phases as a function of the number of processors. In these figures, we plot not only studied ordering policies but also a curve presenting the optimal solution provided by CPLEX solver. In general, the algorithm using the ordering policy achieving the lowest rejection rate has its competitive ratio of 2 or 3, which are rather good results taking into account that our search is not exhaustive compared to the search for optimal solution.

Scenario RANGE is again not presented because the rejection rate of all ordering policies no matter communication phase is close or equal to 0 (in general the rejection rate is less than 1%) due to lower system load. As for depicted scenarios, it is not so straightforward to determine one policy, which always performs well. A reasonable choice is the "Minimum Slack" or "Earliest Phase" during the communication phase and the "Minimum Slack" or "Longest Execution Time" during the no-communication phase. Altogether, the "Minimum Slack" policy performs well regardless of communication phase. Nevertheless, the rejection rate of OneOff&Cyclic is in general higher than the one of OneOff.

6.2.3 Comparison of Different Scenarios

The performance of a given ordering policy are influenced by the system load and the task proportions. The influence of the former factor is illustrated by Scenario RANGE, which has much lower (or none) rejection rate than other two scenarios. The impact of the latter factor is demonstrated by the difference in the rejection rate for Scenarios APSS and APSS-modified. For several ordering policies, the rejection rate is higher during the no-communication phase than during the communication one despite the fact that there are less tasks during the no-communication phase. Actually, there are 29.4% double tasks during the no-communication phase against 4.2% double tasks during the communication phase. To illustrate this difference, Figure 12 shows the proportion of simple and double tasks against the rejection rate for Scenario APSS as a function of the number of processors for both communication phases when OneOff using the "Earliest Deadline" policy is put into practice.



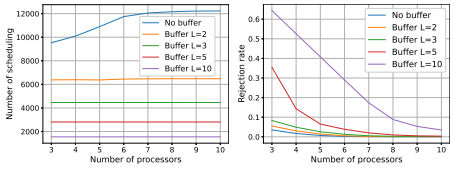
(a) Communication phase (b) No-communication phase

Figure 12: Proportion of simple and double tasks against the rejection rate as a function of the number of processors (OneOff using the "Earliest Deadline" policy; Scenario APSS)

In order not to oversize the system, it is useless to consider more than 6 processors because, when an ordering policy is well chosen, no task is rejected.

6.3 Evaluation of the Buffer for OneOff

In Section 4.3, we presented a method to reduce the number of scheduling searches. This method is now assessed in terms of the number of scheduling searches and rejection rate depicted in Figure 13 for Scenario APSS during the communication phase. The algorithm makes use of the "Earliest Deadline" policy. We consider that the slack constants β and γ are equal and set at 2. The buffer length L varies in the range from 1 to 10. When $L = 1$, the proposed method is not considered and the algorithm runs as OneOff.



(a) Number of scheduling searches (b) Rejection rate

Figure 13: System performance as a function of the number of processors (OneOff using the "Earliest Deadline" policy; Scenario APSS; communication phase)

Figure 13a shows that the use of buffer is helpful to reduce the number of scheduling searches. If we take a 6-processor system as an example, the buffer length $L = 2$ reduces the number of scheduling searches by 45%. Longer buffers reduce it even more but at the cost of higher rejection rate, which is depicted in Figure 13b. The longer the buffer, the more tasks rejected but when the buffer length $L = 2$ or $L = 3$, the increase in the rejection rate is negligible. In general, when a task is put into the buffer, processors may be idle while it is in the buffer. Later, the processors may not be able to accommodate all tasks, which need to be scheduled.

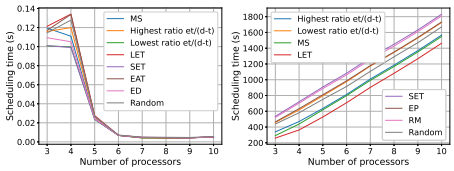
Next, the detailed analysis (figures not presented in this paper) was carried out in order to find values of the buffer length L and the slack constants β and γ. We found out that these values mainly depend on an application. In general, if the buffer is shorter, there are more scheduling searches because the buffer cannot accommodate more tasks. By contrast, if it is longer, there are several tasks in the buffer having short slack so the buffer needs to be emptied.

Although the idea to set a limitation on the number of scheduling searches is interesting (for example, when the buffer length $L = 2$ or $L = 3$, the increase in the rejection rate is negligible but the decrease in the number of scheduling is significant), the user should be aware of two possible issues. Firstly, a limitation increases the rejection rate, which is the metric one usually wants to minimise. Secondly, when setting the values of β and γ, the algorithm is not general any

more because the choice of the values would be probably application-dependent.

6.4 Comparison of Scheduling Time

In this section, we compare the scheduling time of OneOff and OneOff&Cyclic when the algorithm executes on 2 Intel Xeon Processors E5640 running up to 2.67 GHz. In order to show the data corresponding to the rejection rate presented in Section 6.2, we do not make use of the buffer. First, we will analyse Scenario APSS and then Scenario APSS-modified. Scenario RANGE is not presented because the results of OneOff&Cyclic are qualitatively similar to the ones of Scenario APSS. As for the results of OneOff, there are several variations since the task queue length does not have significant differences for different ordering policies as for Scenario APSS.



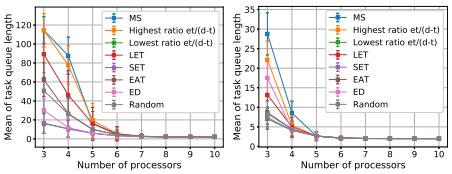
(a) OneOff (b) OneOff&Cyclic

Figure 14: Scheduling time as a function of the number of processors (Scenario APSS; communication phase)

Figure 14 represents the scheduling time of **Scenario APSS** for OneOff and OneOff&Cyclic during the communication phase as a function of the number of processors. The scheduling time during the no-communication phase is qualitatively similar to the one in Figure 14 but approximately 4 times shorter for OneOff&Cyclic and 2 times shorter for OneOff (when there is less than 5 processors). The communication phase takes more time to find a schedule than the no-communication phase because there are more tasks.

Moreover, there is no significant difference among ordering policies for OneOff while there is one for OneOff&Cyclic. The ordering policies that achieve the lowest scheduling time for OneOff are the "Shortest Execution Time", "Lowest ratio of $et/(d-t)$" and "Earliest Deadline". As for OneOff&Cyclic, we point out the "Longest Execution Time", "Minimum Slack" and "Highest ratio of $et/(d-t)$" techniques as the best ordering policies and the "Shortest Execution Time" and "Rate Monotonic" techniques as the worst ones in terms of the scheduling time. To demonstrate the gap, we consider a 3-processor system during the communication phase: the "Shortest Execution Time" technique needs 536 s, which is roughly double than the "Longest Execution Time" technique requiring 260 s.

The scheduling time is related to the algorithm complexity, which is defined in Sections 4.2 and 4.4 for OneOff and OneOff&Cyclic, respectively. One of the terms standing for the complexity is the number of tasks in the task queue. To show the trend of the task queue length, Figure 15 depicts the mean value



(a) Communication phase (b) No-communication phase

Figure 15: Mean value of the task queue length with standard deviations as a function of the number of processors (OneOff; Scenario APSS)

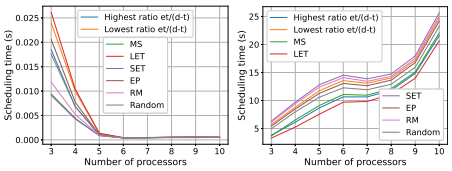
of the task queue length with standard deviations during both communication phases for OneOff and Scenario APSS. We notice that the higher the number of processors, the shorter the task queue and that ordering policies have significant differences in the number of tasks in the task queue when a system has a low number of processors. As an example, we compare the "Shortest Execution Time" technique with the "Longest Execution Time" technique. The former technique has shorter task queue than the latter one because shorter tasks are scheduled first, which implies more scheduling triggers and subsequently more searches for a new schedule.

Consequently, the scheduling time of OneOff decreases with the higher number of processors because the task queue is shorter owing to more scheduling triggers. Regarding the scheduling time of OneOff&Cyclic, it raises when the number of processors increases even though the number of tasks is almost constant (the set of periodic tasks remains the same for a given phase and there is only one arrival of aperiodic task). The increase is due to more possibilities to be tested when a system has more processors.

Nonetheless, as the results are based on simulations (since real experiments are not easily feasible), the scheduling time in our experiments does not significantly change as the task queue length could foresee. This difference is due to the additional complexity related to our simulation framework (handling of arrays in time standing for schedules on processors), which will not be present in reality and the real scheduling time will be shorter.

Finally, the scheduling time of OneOff&Cyclic is roughly 5 orders of magnitude greater than the one of OneOff. This huge gap is mainly due to the significant difference in task periods: between 500 ms and 60000 ms. To better evaluate this impact on scheduling time, we modified Scenario APSS to Scenario APSS-modified, as described in Section 5.

Figure 16 represents the scheduling time of **Scenario APSS-modified** for OneOff and OneOff&Cyclic during the communication phase as a function of the number of processors. The trend of scheduling time during the no-communication phase is again similar to the one in Figure 16 and the values are divided by a number within the range from 5 to 10 for OneOff&Cyclic and by 2 for OneOff (when a system has less than 6 processors).



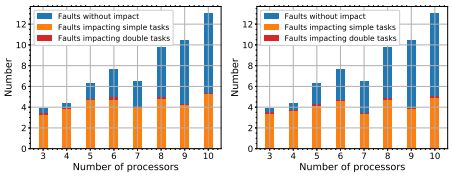
(a) OneOff (b) OneOff&Cyclic

Figure 16: Scheduling time as a function of the number of processors (Scenario APSS-modified; communication phase)

The scheduling time of OneOff&Cyclic is roughly 3 orders of magnitude greater than the one of OneOff. We conclude that the idea to reduce the substantial difference in the task periods accelerates the scheduling time. Therefore, we suggest to teams building CubeSats to avoid tasks with very short and very long periods to be scheduled together.

6.5 Fault Injection

In this section, we evaluate the fault tolerance of both algorithms for Scenario APSS. We consider the "Earliest Deadline" policy for OneOff and the "Minimum Slack" policy for OneOff&Cyclic.



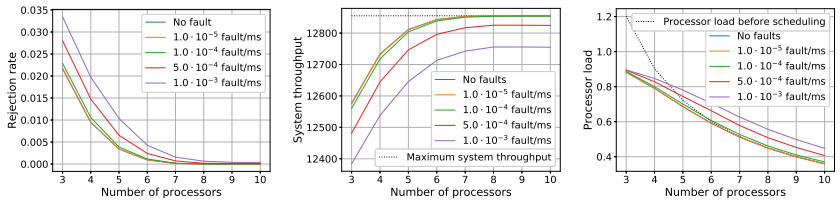
(a) Communication phase (b) No-communication phase

Figure 17: Number of faults (injected with fault rate $1 \cdot 10^{-5}$ fault/ms) and their proportion respectively impacting simple and double tasks (OneOff using the "Earliest Deadline" policy; Scenario APSS)

Figure 17 represents the number of faults (injected with fault rate $1 \cdot 10^{-5}$ fault/ms, which corresponds to the worst estimated fault rate in the real space environment [24]) and their proportion respectively impacting simple and double tasks as a function of the number of processors. Albeit only values for OneOff are shown, the ones for OneOff&Cyclic are similar. We remind the reader that presented results were computed as an average of 20 simulations and thereby they may not be integers.

The number of impacted tasks remains almost constant and there is no significant difference between two algorithms nor between communication phases. Furthermore, double tasks are rarely impacted, which is due to their shorter

execution time when compared with simple tasks. We also studied other fault rates (graphs not shown in this paper). As expected, the higher the fault rate, the more faults. Nonetheless, the proportions of impacted simple and double tasks remain the same.



(a) Rejection rate (b) System throughput (c) Processor load

Figure 18: System performance at different fault injection rates as a function of the number of processors (OneOff using the "Earliest Deadline" policy; Scenario APSS; communication phase)

Figure 18 depicts the rejection rate, system throughput and processor load for communication phase as a function of the number of processors. Qualitatively similar results were obtained for OneOff during the no-communication phase and for both phases of OneOff&Cyclic. The figure representing the system throughput includes a black dashed line corresponding to the case when no task is rejected and all tasks are correctly executed. Regarding the figure plotting the processor load, it also shows a black dashed line, which denotes the maximum processor load. This maximum value is computed as the sum of all execution times of tasks at the input (in a fault-free environment) divided by the simulation duration.

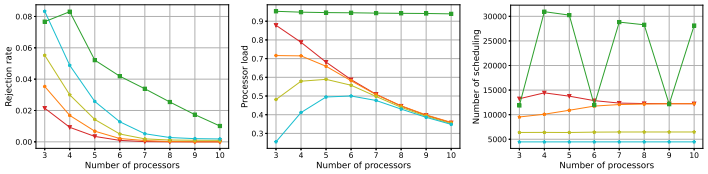
The higher the number of processors, the lower the rejection rate and the higher the system throughput because the number of tasks to be executed aboard the CubeSat is always the same for a given phase. The rejection rate stands for the schedulability as described in Section 4, i.e. if a fault occurs during a PC execution, the corresponding backup copy is scheduled. Nonetheless, a backup copy may be impacted by a fault too. Since such a task was not correctly executed, it does not contribute to the system throughput.

Moreover, the higher the fault rate, the higher the rejection rate and processor load and the lower the system throughput because the backup copies are executed and not deallocated, which increases the system load. The processor load can be even higher than its maximum theoretical value computed in a fault-free environment because the backup copies need to be executed if a fault occurs during the execution of the corresponding primary copies. Furthermore, we notice that the studied metrics do not change significantly up to $1 \cdot 10^{-4}$ fault/ms, which is higher than the worst estimated fault rate in the real space environment (10^{-5} fault/ms [24]). The same conclusions were made for other two scenarios (RANGE and APSS-modified) as well.

Although only transient faults were studied, the CubeSat performance after an occurrence of permanent fault can be foreseen. If a permanent fault occurs

causing a processor failure, a CubeSat loses one processor. Since we consider that there are no dedicated processor(s) to each CubeSat system, any processor can execute any task, as described in Section 1. Therefore, a permanent fault would not be a problem because there are still enough computational resources, which is an advantage of the proposed solution. Furthermore, the fault rate of permanent faults is lower than the one of transient faults. For example, the fault rate of permanent hardware faults in a multicore chip is $10^{-5}/h$ and the fault rate of non-permanent hardware faults in each core during non-bursty period is $10^{-4}/h$ [24].

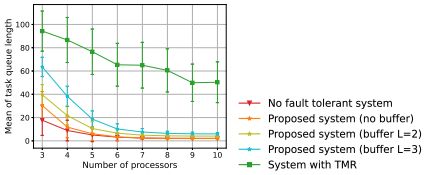
6.6 Comparison of the Proposed Solution with No-Fault Tolerant and TMR Systems



(a) Rejection rate

(b) Processor load

(c) Number of scheduling searches



(d) Mean value of the task queue length with standard deviations

Figure 19: System performance for three systems with different level of fault tolerance as a function of the number of processors (OneOff using the "Earliest Deadline" policy; Scenario APSS; communication phase)

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). 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, as explained in Section 3, and schedules backup copies only if a fault occurs.

Figure 19 depicts the rejection rate, processor load, number of scheduling searches and mean value of the task queue length with standard deviations as a function of the number of processors for the workload aboard APSS CubeSat during the communication phase.

While the rejection rate of our proposed system (without or with a short buffer $L = 2$) is only slightly higher when compared to the rejection rate of a system without any redundancy, the rejection rate of a system based on the TMR is significantly higher. For example, a 6-processor system rejects 0.093% (system without any redundancy), 0.21% (proposed system without buffer), 0.50% (proposed system with buffer $L = 2$) and 4.05% (system with TMR). The thorough analysis shows (graph not presented in this paper) that, in order to schedule (almost) all tasks aboard APSS CubeSat, the system using the TMR should have at least 12 processors, i.e. 6 processors more than the system taking advantage of our proposed solution for the same reliability level.

When the rejection rate is 0, Figure 19b shows that our proposed solutions without and with a short buffer has similar values of the processor load as the system without any redundancy. The system using the TMR has higher processor load than the other systems because all tasks have three task copies. For instance, the processor load of 8-processor system is 44.6% (system without any redundancy), 44.8% (proposed system without buffer), 43.9% (proposed system with buffer $L = 2$) and 94.5% (system with TMR). The analysis for more than 10 processors (graph not presented in this paper) shows that the processor load of the system using the TMR decreases and that it is roughly three times higher than a system without any redundancy. Since the power consumption is related to the system load, the power consumption of the system with TMR is higher and the one of the system making use of our solution is similar to the system without any redundancy.

Besides higher processor load, the system based on TMR presents other inconveniences. Its number of scheduling searches is higher when compared to other systems, as depicted in Figure 19c. For example, the number of scheduling searches for a 8-processor system is 12258 (system without any redundancy), 12161 (proposed system without buffer), 6477 (proposed system with buffer $L = 2$) and 28478 (system with TMR). The reason why the number of scheduling searches decreases when the number of processors is a multiple of three, which corresponds to the number of task copies, remains unknown. Therefore, the scheduler of the system with TMR carries out many searches for a new schedule which is not very suitable for embedded systems aboard CubeSats due to limited power resources. Our proposed solutions, especially the ones making use of the buffer, are well suited to such applications.

Furthermore, the tasks queue of the system with TMR is longer than for other systems, as shown in Figure 19d. Since the task queue length is related to the algorithm complexity, as explained in Section 6.4, the complexity of the system based on the TMR is higher when compared to other systems. Again, although the task queue length of our proposed systems is slightly higher than for a system without any redundancy, our proposed system offers an interesting solution subject to trade-off between the number of scheduling searches and the task queue length.

7 Conclusion

This paper evaluated the performance of two online algorithms meant for CubeSats, which operate in the harsh space environment and are vulnerable to faults. To make CubeSats fault tolerant, these algorithms schedule all tasks aboard the CubeSat, detect faults and take appropriate measures in order to deliver correct results.

While the first algorithm (called OneOff) considers all tasks as aperiodic tasks, the second one (named OneOff&Cyclic) distinguishes aperiodic and periodic tasks when searching for a new schedule. Each algorithm can use different ordering policies to sort a task queue. The presented results based on two real CubeSat scenarios show that it is useless to consider systems with more than six processors and that OneOff performs better than OneOff&Cyclic in terms of the rejection rate and the scheduling time. OneOff&Cyclic can be more efficient in applications where there are only a few changes in the set of periodic tasks. Therefore, we suggest that teams, which design their CubeSats gathering all processors together on one board, put into practice rather OneOff and use a short buffer to reduce the number of scheduling searches.

Last but not least, the results show that fault rates up to $1 \cdot 10^{-4}$ fault/ms, which is higher than the worst estimated fault rate in the real space environment, have minimal impact on performance of both algorithms. Moreover, our proposed solution provides the same reliability level as a system with triple modular redundancy (TMR) but without significantly increasing processor load. Therefore, the power consumption of our system remains similar to the system without any redundancy and is significantly lower than for a system based on the TMR. This is achieved by distinguishing the simple and double tasks requiring a re-execution only if a fault occurs.

All in all, the results show that a proposed solution based on a multiprocessor system is well suited to CubeSats in order to improve their fault tolerance taking into account their constraints.

As our future work, we are about to further evaluate energy constraints, which play an important role to ensure real-time execution of tasks because a CubeSat spends one third of its orbit in the eclipse with limited power supply.

Acknowledgments

The authors would like to thank the Auckland Program for Space Systems (APSS) and Space Systems Design Lab (SSDL) for sharing their CubeSat data. The authors would like to express their thanks to Dr. Bertrand Granado (Sorbonne Université, CNRS, LIP6, Paris, France) for valuable inputs to this research.

References

[1] Arizona State University, Phoenix PDR. Presentation on March 24,

2017 at AMSAT-UK Colloquium 2014, 2017. http://phxcubesat.asu.edu/sites/default/files/general/phoenix_pdr_part_2_1.pdf.

[2] D. Burlyaev, System-level Fault-Tolerance Analysis of Small Satellite On-Board Computers, Master's thesis, Delf University of Technology, 2012. https://repository.tudelft.nl/islandora/object/uuid:b467aa94-76d9-4425-8ed2-4f9a0121d04a?collection=education.

[3] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applicat Springer, 2011.

[4] J. P. Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende, Multi-Core Devices for Safety-Critical Systems: A Survey, in ACM Computing Surveys, vol. 53, New York, NY, USA, 2020, Association for Computing Machinery.

[5] L.-W. Chen, T.-C. Huang, and J.-C. Juang, Implementation of the Fault Tolerance Module in PHOENIX CubeSat. Presentation at 10th IAA Symposium on Small Satellites for Earth Observation, 2015. https://www.dlr.de/iaa.symp/Portaldata/49/Resources/dokumente/archiv10/pdf/0604_IAA-Li-Wei-Chen.pdf.

[6] T. B. Clausen et al., Designing On Board Computer and Payload for the AAU CubeSat. http://www.crn.inpe.br/conasat1/projetos_cubesat/projetos/AAUSAT-AalborgUniversity-Denmark/AAUSAT-OBC-report.pdf.

[7] C. S. U. de Grenoble, ATISE project: Auroral Thermosphere Ionosphere Spectrometer Experiment. https://www.csug.fr/main-menu/projects/atise-project/.

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

[9] P. Dobiáš, E. Casseau, and O. Sinnen, Fault-Tolerant Online Scheduling Algorithms for CubeSats, in 11th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures / 9th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM'20), 2020, pp. 1–6.

[10] G. Dósa and Y. He, Semi-Online Algorithms for Parallel Machine Scheduling Problems, in Computing, vol. 72, Jun 2004, pp. 355–363.

[11] E. Dubrova, Fault-Tolerant Design, Springer, 2013.

[12] Erik Kulu, Nanosats Database. https://www.nanosats.eu/.

[13] A. Erlank and C. Bridges, Reliability Analysis of Multicellular System Architectures for Low-Cost Sa in Acta Astronautica, vol. 147, 2018, pp. 183–194.

[14] A. O. Erlank and C. P. Bridges, Satellite Stem Cells: The Benefits & Overheads of Reliable, Multicellular architectures, in 2017 IEEE Aerospace Conference, March 2017, pp. 1–12.

[15] D. Geeroms et al., ARDUSAT, an Arduino-Based CubeSat Providing Students with the Opportunity t in 22nd ESA Symposium on European Rocket and Balloon Programmes and Related Research, vol. 730 of ESA Special Publication, Sep 2015, p. 643. https://ui.adsabs.harvard.edu/abs/2015ESASP.730..643G.

[16] 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, 1997, pp. 272–284.

[17] H. Kellerer, V. Kotov, M. G. Speranza, and Z. Tuza, Semi On-line Algorithms for the Partition Problem, in Operations Research Letters, vol. 21, 1997, pp. 235–242.

[18] K. A. LaBel, Radiation Effects on Electronics 101: Simple Concepts and New Challenges. Presentation at NASA Electronic Parts and Packaging (NEPP) Webex Presentation, 2004. https://nepp.nasa.gov/docuploads/392333B0-7A48-4A04-A3A72B0B1DD73343/Rad_Effects_101_WebEx.pdf.

[19] K. Laizans et al., Design of the Fault Tolerant Command and Data Handling Subsystem for ESTCube in Proceedings of the Estonian Academy of Sciences, 2014, pp. 222–231.

[20] M. Langer, Reliability Assessment and Reliability Prediction of CubeSats through System Level Testing PhD thesis, Technical University of Munich, 2018. https://mediatum.ub.tum.de/?id=1446237.

[21] J. Mei et al., Fault-Tolerant Dynamic Rescheduling for Heterogeneous Computing Systems, in Journal of Grid Computing, vol. 13, 2015, pp. 507–525.

[22] NASA CubeSat Launch Initiative, CubeSat 101: Basic Concepts and Processes for First-Time CubeSat Developers, 2017. https://www.nasa.gov/sites/default/files/atoms/files/nasa_csli_cubesat_101_508.pdf.

[23] National Aeronautics and Space Administration (NASA), What are SmallSats and CubeSats?, 2019. https://www.nasa.gov/content/what-are-smallsats-and-cubesats.

[24] R. M. Pathan, Real-Time Scheduling Algorithm for Safety-Critical Systems on Faulty Multicore Enviro in Real-Time Systems, vol. 53, 2017, pp. 45–81.

[25] 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.

[26] R. Velazco, D. McMorrow, and J. Estela, eds., Radiation Effects on Integrated Circuits and Systems for Space Applications, Springer, 2019.

[27] S. Wang et al., A Reliability-aware Task Scheduling Algorithm Based on Replication on Heterogeneous in Journal of Grid Computing, vol. 15, 03 2017, pp. 23–39.

[28] Warsaw University of Technology, PW-SAT 2 Preliminary Requirements Review: On-Board Computer, 2014. https://pw-sat.pl/wp-content/uploads/2014/07/PW-Sat2-A-04.00-OBC-PRR-EN-v1.1.pdf.

[29] Q. Zheng, B. Veeravalli, and C.-K. Tham, On the Design of Fault-Tolerant Scheduling Strategies Using Primary-Backup Approach for Computatio in IEEE Transactions on Computers, vol. 58, 2009, pp. 380–393.