Online Fault Tolerant Task Scheduling for Real-Time Multiprocessor Embedded Systems

Petr Dobiáš

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1
COMUE UNIVERSITÉ BRETAGNE LOIRE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Petr DOBIÁŠ

Contribution à l'ordonnancement dynamique, tolérant aux fautes, de tâches pour les systèmes embarqués temps-réel multiprocesseurs

Thèse présentée et soutenue à Lannion, le 2 octobre 2020
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Alberto BOSIO	Professeur des Universités, Ecole Centrale de Lyon, France
Arnaud VIRAZEL	Maître de Conférences, Université de Montpellier, France

Composition du Jury :

Président :	Bertrand GRANADO	Professeur des Universités, Sorbonne Université, France
Examinateurs :	Alberto BOSIO	Professeur des Universités, Ecole Centrale de Lyon, France
	Maryline CHETTO	Professeur des Universités, Université de Nantes, France
	Daniel CHILLET	Professeur des Universités, Université de Rennes 1, France
	Oliver SINNEN	Associate Professor, University of Auckland, Nouvelle Zélande
	Arnaud VIRAZEL	Maître de Conférences, Université de Montpellier, France
Dir. de thèse :	Emmanuel CASSEAU	Professeur des Universités, Université de Rennes 1, France

Résumé

La thèse se focalise sur le placement et l'ordonnancement dynamique des tâches sur les systèmes embarqués multiprocesseurs pour améliorer leur fiabilité tout en tenant compte des contraintes telles que le temps réel ou l'énergie. Les performances du système sont principalement évaluées par le nombre de tâches rejetées, la complexité de l'algorithme et donc sa durée d'exécution et la résilience estimée en injectant les fautes. Les contributions de la recherche sont dans les deux domaines suivants : l'approche d'ordonnancement dite de « primary/backup » et la fiabilité des petits satellites appelés CubeSats.

Description de l'approche de « primary/backup »

L'approche de « primary et backup » (approche PB) considère que chaque tâche a deux copies identiques pour rendre le système tolérant aux fautes [61]. Ces copies sont placées sur deux processeurs différents entre le temps d'arrivée de la tâche et sa date limite d'exécution. La première copie nommée copie de « primary » est placée le plus tôt possible tandis que la deuxième copie appelée copie de « backup » est positionnée le plus tard possible. Pour améliorer l'ordonnancement, les copies de « backup » peuvent être chevauchées entre elles ou désallouées si les exécutions de leurs copies de « primary » respectives sont correctes. D'autres heuristiques pour l'approche PB ont été déjà présentées [61, 103, 144, 155]. Les fautes sont détectées par un mécanisme de détection qui signale leur occurrence.

Contributions à l'approche de « primary/backup »

Le but est de proposer des heuristiques subtiles pour réduire la durée d'exécution (mesurée à l'aide du nombre de comparaisons) de l'algorithme d'ordonnancement tout en évitant la détérioration des performances du système (évaluées par exemple par le taux de réjection, i.e. le nombre de tâches rejetées par rapport au nombre total de tâches). Les contributions à l'approche PB sont les suivantes :

- l'évaluation de la surcharge de cette approche ;
- la proposition d'une nouvelle stratégie d'allocation des processeurs qu'on nomme la « recherche jusqu'à la première solution trouvée – créneau après créneau » (FFSS SbS) et qu'on compare avec d'autres stratégies déjà existantes ;
- la proposition de trois nouvelles heuristiques : (i) la méthode de *limitation du nombre de comparaisons*, (ii) la méthode de *limitation des fenêtres d'ordonnancement* délimitant le temps pendant lequel une copie peut être placée et (iii) la méthode de *plusieurs essais d'ordonnancement* ;
- l'évaluation des performances de l'algorithme, en particulier en termes de nombre de comparaisons par tâche et de taux de réjection, y compris avec l'injection des fautes ;
- la formulation mathématique du problème et la comparaison des résultats avec la solution optimale délivrée par le solveur CPLEX ;
- l'adaptation des algorithmes proposés ci-dessus pour des tâches indépendantes afin de placer des tâches dépendantes.

Analyses des résultats de l'approche de « primary/backup »

Les analyses des résultats pour les tâches indépendantes ont permis de conclure les points suivants.

Le taux de réjection du système autorisant le chevauchement des copies de « backup » est réduit par rapport au système sans aucune technique particulière (par exemple de 14% pour un système avec 14 processeurs). En cas de la désallocation des copies de « backup », il est réduit encore davantage (par

exemple de 75% pour un système avec 14 processeurs). De plus, les résultats montrent que les techniques de chevauchement et de désallocation des copies de « backup » fonctionnent bien ensemble.

Le surcoût de l'approche PB qui place deux copies de la même tâche (même si la copie de « backup » peut être désallouée) a été également évalué. Quand le nombre de processeurs augmente, le nombre de comparaisons par tâche pour trouver une place pour ses copies augmente également et la différence du nombre de comparaisons entre les systèmes sans et avec approche PB devient plus importante. Néanmoins, comme il y a plus de comparaisons effectuées, la probabilité de placer une tâche augmente et donc le taux de réjection du système tolérant aux fautes diminue et se rapproche de celui du système non-tolérant.

Ensuite, on compare les trois stratégies d'allocation des processeurs : la « recherche exhaustive » (ES), la « recherche jusqu'à la première solution trouvée – processeur après processeur » (FFSS PbP) et la « recherche jusqu'à la première solution trouvée – créneau après créneau » (FFSS SbS). L'ES a le taux de réjection le plus bas parmi toutes les stratégies mais ses nombres moyen et maximal de comparaisons par tâche sont au contraire les plus élevés. La méthode FFSS SbS est un bon compromis. Par exemple, le taux de réjection de FFSS SbS est de 12% plus élevé que celui d'ES pour un système avec 14 processeurs et son nombre maximal de comparaisons par tâche est considérablement inférieur par rapport à celui de FFSS PbP (29% pour un système avec 14 processeurs) et à celui d'ES (41% pour un système avec 14 processeurs). De plus, en comparant l'algorithme basé sur FFSS SbS à la solution optimale obtenue par un solveur CPLEX, on trouve qu'il est 2-compétitive.

Puis, deux techniques pour parcourir les processeurs sont étudiées : la « recherche basée sur les créneaux disponibles » (FSST) et la « recherche basée sur les débuts et les fins des copies déjà placées » (BSST). La méthode BSST + ES et la méthode FSST + ES ont des nombres similaires de tâches rejetées et BSST a besoin plus que deux fois plus de comparaisons que la FSST. Ainsi, BSST n'est pas une technique à choisir en termes de durée d'exécution de l'algorithme.

Après les analyses des stratégies d'allocation des processeurs et des techniques pour parcourir les processeurs, on s'intéresse aux performances des heuristiques qu'on propose.

La méthode de *limitation du nombre de comparaisons* montre que la définition du seuil permet de réduire le nombre maximal des comparaisons. Par exemple, si ce seuil pour les copies de « primary » est fixé à $P/2$ comparaisons (où P est le nombre de processeurs) et celui pour les copies de « backup » est égal à 5 comparaisons, les nombres maximal et moyen des comparaisons par tâche respectivement diminuent de 62% et 34% tandis que le taux de réjection augmente seulement de 1,5% en comparant avec l'approche PB sans cette méthode.

La méthode de *limitation des fenêtres d'ordonnancement* est aussi efficace pour réduire le nombre de comparaisons sans aggraver les performances du système. Un compromis raisonnable entre le nombre de comparaisons et le taux de réjection est obtenu pour la fraction de la fenêtre de tâche égale à 0,5 ou 0,6.

La troisième heuristique proposée, *plusieurs essais d'ordonnancement*, vise à abaisser le taux de réjection des tâches. Les résultats montrent qu'il est inutile de réaliser plus que deux essais car, quand le nombre d'essais augmente, le taux de réjection ne diminue que marginalement et le nombre de comparaisons par tâche augmente assez vite. Un bon compromis entre ces deux métriques est obtenu pour deux essais ayant lieu à 33% de la fenêtre de tâche. Dans ce cas-là, le taux de réjection décroît de 6,2%.

En comparant les heuristiques et leurs combinaisons en termes de taux de réjection et du nombre de comparaisons, on trouve que les meilleurs résultats sont obtenus pour : (i) la méthode de limitation du nombre de comparaisons utilisant deux essais à 33% de la fenêtre de tâche et (ii) la méthode de limitation du nombre de comparaisons. Dans le premier cas mentionné, le nombre de comparaisons diminue considérablement (valeur moyenne : 23% ; valeur maximale : 67%) et le taux de réjection est réduit de 4% par rapport à l'approche PB sans aucune technique d'amélioration.

Pour évaluer les performances en présence des fautes, l'algorithme proposé a été testé par l'injection des fautes. On a constaté que les injections des fautes allant jusqu'à $1 \cdot 10^{-3}$ fautes par processeurs/ms ont un impact minimal. Comme cette valeur est supérieure à la valeur estimée dans les conditions standard ($2 \cdot 10^{-9}$ fautes par processeurs/ms [47]) et à celle dans les conditions rudes ($1 \cdot 10^{-5}$ fautes par processeurs/ms [118]), l'algorithme peut donc être implémenté dans les systèmes exposés à l'environnement hostile.

Afin de prolonger l'étude sur l'approche PB, l'algorithme proposé a été modifié pour gérer les tâches

dépendantes modélisées par les graphes orientés acycliques (DAG). Les deux techniques pour parcourir les processeurs (FSST et BSST) combinées à trois stratégies d'allocation des processeurs (ES, FFSS PbP et FFSS SbS) ont été de nouveau comparées. Le nombre de comparaisons par DAG pour BSST + ES est considérablement plus élevé que pour les deux autres techniques (FSST + FFSS PbP et FSST + FFSS SbS) ce qui est dû au type de la recherche : exhaustive ou pas. Bien que la FFSS SbS et la FFSS PbP aient un taux de réjection similaire, la FFSS SbS nécessite plus de comparaisons. La méthode maximisant le chevauchement entre les copies de « backup » (BSST + ES maxOverload) a les meilleurs résultats en termes de taux de réjection mais au détriment de la durée d'exécution de l'algorithme, sauf pour les systèmes ayant peu de processeurs. L'injection des fautes a montré que l'algorithme proposé fonctionne bien même avec les taux d'injection des fautes supérieurs aux valeurs réelles dans les conditions difficiles.

Description des CubeSats

Les CubeSats sont les petits satellites envoyés dans l'orbite basse de la Terre avec des missions scientifiques. Leurs popularité augmente grâce à la standardisation qui réduit le budget et le temps de développement [52]. Ils sont composés d'un ou plusieurs cubes d'arête de $10\,cm$ et de poids maximal de $1,3\,kg$ [108]. À bord, il y a en général plusieurs systèmes électroniques systèmes, comme l'ordinateur de bord, le système de la détermination d'attitude et de contrôle ou le système lié à la mission (partie scientifique).

Les CubeSats sont exposés aux particules chargées et aux radiations qui causent des effets singuliers, par exemple « Single Event Upset » (SEU) et des effets de dose, comme « Total Ionizing Dose » (TID) [89]. Il est donc nécessaire de concevoir des CubeSats plus robustes. Les méthodes de robustification ne sont pas de manière générale utilisées en raison des contraintes budgétaire, du temps de conception ou de l'espace disponible [55]. Par exemple, il y a 43% de CubeSats qui ne mettent pas en œuvre la redondance, technique classique au niveau matériel [54, 90]. En raison de contraintes spatiales, il est préférable d'utiliser les méthodes logicielles, comme les watchdogs ou les techniques protégeant les données [3, 36, 38].

Contributions aux CubeSats

Pour améliorer la fiabilité des CubeSats, on propose de regrouper tous les processeurs à bord sur une même carte ayant un seul système intégré. Même si cette modification peut paraître importante pour les CubeSats actuels, elle a été déjà réalisée avec succès à bord d'ArduSat avec 17 processeurs [58]. Ainsi, il sera plus facile de protéger les processeurs, par exemple en utilisant un blindage contre les radiations [30], et d'augmenter les chances de bon déroulement de la mission car si un processeur est défectueux, d'autres processeurs qui ne sont pas dédiés à un système donné (comme c'est le cas dans les CubeSats actuels) continuent à fonctionner.

Dans ce cadre-là, on a développé des algorithmes qui placent toutes les tâches (périodiques, sporadiques et apériodiques) à bord de CubeSat, détectent des fautes et prennent des mesures pour délivrer des résultats corrects. L'objectif est de minimiser le nombre de tâches rejetées en respectant les contraintes temporelles, énergétiques et la fiabilité. Ces algorithmes sont exécutés dynamiquement pour immédiatement réagir. Ils sont principalement dédiés aux CubeSats basés sur les processeurs commerciaux standard qui ne sont pas conçus pour l'utilisation dans l'espace contrairement aux processeurs durcis.

Les contributions dans le domaine des CubeSats sont les suivantes :

— l'évaluation des performances de trois algorithmes d'ordonnancement proposés, dont un tenant compte des contraintes énergétiques, en termes de taux de réjection, de nombre de recherches d'ordonnancement effectuées et de durée d'exécution d'algorithme ;
— la formulation mathématique du problème et la comparaison des résultats avec la solution optimale délivrée par le solveur CPLEX ;
— l'évaluation de la durée du fonctionnement du système en utilisant l'algorithme proposé prenant en compte les contraintes énergétiques ;
— l'injection des fautes et l'analyse de l'impact sur les performances du système ;

— en se basant sur les résultats obtenus, la recommandation du choix de l'algorithme à choisir.

Analyses des résultats des CubeSats

L'algorithme appelé OneOff considère toutes les tâches comme apériodiques et l'algorithme nommé OneOff&Cyclic distingue les tâches périodiques et apériodiques. Tandis que ces deux algorithmes ne tiennent pas compte de contraintes énergétiques, l'algorithme OneOffEnergy les considère. Tous les algorithmes peuvent utiliser différentes stratégies de placement pour ordonner la queue des tâches.

Les performances de OneOff et OneOff&Cyclic ont été étudiés avec trois scénarios, dont deux proviennent de réels CubeSats. Les scénarios diffèrent par la charge du système et le rapport entre les tâches simples et doubles.

Les résultats montrent qu'il est inutile de considérer un système ayant plus de six processeurs car, si un stratégie d'ordonnancement est bien choisi, il n'y a pas de tâche rejetée. Ce choix permet donc d'éviter un système surdimensionné. De manière générale, les stratégies de placement "Earliest Deadline" pour OneOff et "Minimum Slack" pour OneOff&Cyclic minimisent bien la fonction objectif, i.e. le taux de réjection. Elles ont également de bonnes performances en termes de durée de l'ordonnancement.

Même s'il a été trouvé que OneOff&Cyclic fonctionne moins bien que OneOff, ce dernier algorithme peut très bien être utilisé dans d'autres applications avec beaucoup plus de profits (par exemple dans les systèmes embarqués avec les contraintes temporelles sévères) ayant moins de déclencheurs d'ordonnancement (moins de fautes, ou moins des tâches apériodiques ou moins de changements dans l'ensemble des tâches périodiques) que dans les applications étudiées.

Ainsi, les équipes construisant leurs propres CubeSats qui regroupent tous les processeurs sur une seule carte, devraient choisir plutôt OneOff si elles hésitent entre les deux algorithmes ne prenant pas en compte les contraintes énergétiques. Néanmoins, il serait mieux d'implémenter le troisième algorithme OneOffEnergy prenant également en compte les contraintes énergétiques.

OneOffEnergy profite de deux régimes du processeur (Run and Standby) pour réduire la consommation énergétique et fonctionne dans un des trois régimes (normal, safe et critical) suivant le niveau d'énergie disponible dans la batterie. Cet algorithme proposé a été évalué non seulement dans le cas des CubeSats mais aussi pour une autre application ayant des contraintes énergétiques.

Le bilan énergétique établi pour le Scénario APSS montre que la phase de communication requiert une quantité d'énergie non-négligeable en raison de la consommation importante de l'émetteur. Même si cette phase ne dure que 10 minutes ce qui est une durée plutôt courte par rapport à la période orbitale du CubeSat étant de 95 minutes, elle peut épuiser la batterie si un algorithme tenant compte de l'aspect énergétique n'est pas implémenté. Si un tel algorithme est mis en service, il n'y a pas de risque de pénurie d'énergie car l'énergie récupérée est suffisante pour couvrir toutes les dépenses énergétiques.

Pour évaluer davantage les performances de OneOffEnergy, les simulations pour une autre application ayant des contraintes énergétiques ont été réalisées et les résultats entre OneOffEnergy et d'autres algorithmes plus simples ont été comparés.

L'évaluation de l'utilisation du mode Standby montre des économies en énergie non-négligeables. En effet, elles contribuent à la durée de fonctionnement plus longue dans les régimes normal et safe ce qui réduit la réjection automatique des tâches de priorité faible. Même si le système ne fonctionnant qu'en régime normal a un taux de réjection inférieur par rapport au système implémentant OneOffEnergy (par exemple de 19% pour le système composé de six processeurs), la capacité de la batterie ne permet pas le fonctionnement continu. Au contraire, OneOffEnergy choisit le régime de fonctionnement (normal, safe ou critical) suivant le niveau d'énergie dans la batterie, exécute les tâches avec un certain niveau de priorité pour optimiser la consommation énergétique et évite une pénurie d'énergie. Ainsi, l'algorithme proposé présente un compromis raisonnable entre le fonctionnement du système, tel que le nombre de tâches exécutées et leurs priorités, et les contraintes énergétiques.

Finalement, les simulations avec l'injection des fautes ont été réalisées. Les résultats montrent que les trois algorithmes proposés (OneOff, OneOff&Cyclic et OneOffEnergy) fonctionnent bien même en environnement hostile.

ACKNOWLEDGEMENT

The author is first and foremost grateful to Dr. Emmanuel Casseau for support, frequent encouragement and numerous fruitful discussions we had during the development of this work.

I also owe an enormous debt of gratitude to Dr. Oliver Sinnen for his assistance, support and opportunity to spend several months at the Parallel and Reconfigurable Computing Lab (PARC) at the University of Auckland, New Zealand. Our discussions were always stimulating and greatly contributed to progress in my PhD thesis.

I am also very grateful to the research CAIRN team at the laboratory of IRISA and the research team at the Parallel and Reconfigurable Computing Lab in Auckland, New Zealand for their support.

Last but not least, I would like to express many thanks to CubeSat teams, such as Phoenix (Arizona State University, USA), RANGE (Georgia Institute of Technology, USA) or PW-Sat2 (Warsaw University of Technology, Poland) for sharing their data and discussions we had. In particular, I also wish to recognize the members of Auckland Programme for Space Systems (APSS) for initiating me into the CubeSat project.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

ADCS	Attitude Determination and Control System.
ALAP	As Late As Possible.
ASAP	As Soon As Possible.
BC	Backup Copy.
BSST	Boundary Schedule Search Technique.
CDHS	Command and Data Handling System.
COM	COMmunication system.
COTS	Commercial Off-The-Shelf.
CP+NCP	Communication Phase and No-Communication Phase.
CPU	Central Processing Unit.
DAG	Directed Acyclic Graph.
DOA	Dead On Arrival.
DOD	Depth Of Discharge.
DVFS	Dynamic Voltage and Frequency Scaling.
EPS	Electrical Power System.
ES	Exhaustive Search.
FFSS SbS	First Found Solution Search: Slot by Slot.
FFSS PbP	First Found Solution Search: Processor by Processor.
FSST	Free Slot Search Technique.
HPC	High-Performance Computing.
HT	Hyperperiod.
ISS	International Space Station.
LANL	Los Alamos National Laboratory.
LEO	Low Earth Orbit.
LET	Linear Energy Transfer.
MIPS	Million Instructions Per Second.
MTBF	Mean Time Between Faults.
MTTF	Mean Time To Faults.
MTTR	Mean Time To Repair.
NASA	National Aeronautics and Space Administration.
NCP	No-Communication Phase.
OBC	On-Board Computer.
PB	Primary/Backup.
PC	Primary Copy.
RX	Receiver.

SEB Single Event Burnout.
SEE Single Event Effect.
SEFI Single Event Functional Interrupt.
SEGR Single Event Gate Rupture.
SEL Single Event Latch-up.
SEMBE Single Event Multiple Bit Error.
SET Single Event Transient.
SEU Single Event Upset.
SLOC Source Lines Of Codes.

TGFF Task Graph For Free.
TID Total Ionising Dose.
TMR Triple Modular Redundancy.
TPL Targeted Processor Load.
TTNF Time To Next Fault.
TX Transmitter.

INTRODUCTION

Every system component is liable to fail and it will cease to correctly run sooner or later. As a consequence, the system can exhibit a malfunction. There are applications where a system failure can have catastrophic consequences such as advanced driver-assistance systems, air traffic control or medical equipment. In order to deal with this problem, systems should be fault tolerant. It means that such a system is more robust, can tolerate several faults and properly works even if faults occur.

In general, requirements on multiprocessor embedded systems for higher performance and lower energy consumption are increasing so that they might meet demands of more and more complex computations. Moreover, the transistors are scaling down and their operating voltage is getting lower, which goes hand in glove with higher susceptibility to system failure.

Since systems are more vulnerable to faults, the reliability becomes the main concern [105]. There are various methods to provide systems with fault tolerance and the choice of the design depends on a particular application [49, 72, 85]. For multiprocessor embedded systems, one of promising methods makes use of reconfigurable computing and/or redundancy in space or in time. In addition, multiprocessor systems are less vulnerable than a standalone processor because, in case of a processor failure, other processors remain operational.

The focus of this PhD thesis is dual. We first deal with the *primary/backup approach* for failure elimination techniques and then with some aspects of scheduling algorithm design of small satellites called *CubeSats*. In both cases, we are concerned with multiprocessor embedded systems with aim to improve their reliability.

The primary/backup approach is a method of fault tolerant scheduling on multiprocessor embedded systems making use of two task copies: the primary and backup ones [61]. It is a commonly used technique for designing fault tolerant systems owing to its easy application and minimal system overheads. Several additional enhancements [61, 103, 144, 155] to this approach have been already presented but few studies dealing with overall comparisons have been published. Moreover, the resiliency of the primary/backup approach has been discussed in only few studies and with several restrictive assumptions.

CubeSats are small satellites consisting of several processors and subject to strict space and weight constraints [108]. They operate in the harsh space environment, where they are exposed to charged particles and radiation [89]. Since the CubeSat fault tolerance is not always considered, e.g., due to budget or time constraints, their vulnerability to faults can jeopardise the mission [54, 90]. Our aim is to improve the CubeSat reliability. The proposed solution again makes use of an online fault tolerant scheduling on multiprocessor embedded systems. It is 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.

Scope of Research

The scope of the PhD thesis is also dual: the first one is related to the primary/backup approach and the second one is concerned with scheduling algorithms for CubeSats to improve their reliability.

Regarding the primary/backup approach, our main objective is to choose enhancing method(s), which significantly reduce(s) the algorithm run-time without worsening system performances when online scheduling tasks on embedded systems. The scope of research meant for the scheduling of independent tasks is as follows:

— Evaluation of the overheads of the primary/backup approach;

— Introduction of a new processor allocation policy (called *first found solution search: slot by slot*) and its comparison with already existing processor allocation policies;
— Introduction and analysis of three new enhancing techniques based on the primary/backup approach: (i) the method of *restricted scheduling windows* within which the primary and backup copies can be scheduled, (ii) the method of *limitation on the number of comparisons*, accounting for the algorithm run-time, when scheduling a task on a system, and (iii) the method of *several scheduling attempts*;
— Discussion of the trade-off between the algorithm run-time (measured by the number of comparisons to find a free slot) and system performances (assessed by the rejection rate, i.e. the ratio of rejected tasks to all arriving tasks);
— Mathematical programming formulation of the scheduling problem and comparison of our results with the optimal solution delivered by CPLEX solver;
— Assessment of the fault tolerance of the primary/backup approach when scheduling independent tasks.

The scope designed for dependent tasks is as reads:
— Adaptations of the scheduling algorithms of independent tasks for dependent ones;
— Evaluation of the scheduling algorithms in terms of their performances and compare them with the already known ones for the dependent tasks;
— The fault tolerance analysis for scheduling dependent tasks.

Regarding CubeSats, our aim is to minimise the number of rejected tasks subject to real-time, reliability and energy constraints. The scope of research related to CubeSats is as follows:
— Assessment of performances of three proposed algorithms in terms of the rejection rate (which again represents the ratio of rejected tasks to all arriving tasks), the number of scheduling searches and the scheduling time for different scenarios;
— Mathematical programming formulation of the scheduling problem; whenever possible we compare the results to the optimal solution provided by CPLEX solver;
— Evaluation of the devised energy-aware algorithm in terms of the system operation and the energy consumption;
— Analyses of the presented algorithms regarding their treatment of faults;
— Based on performances of these algorithms, the suggestion which algorithm should be used on board of the CubeSat.

Paper Organisation

The thesis is organised as follows.

To make the reader familiar with the context of the PhD thesis, Chapter 1 presents an overview of several topics closely related to the carried out research and gives several definitions to introduce main terms, which will be used throughout the thesis. This chapter sums up system, algorithm and task classifications. Then, it summarises fault models, based on either the Markov model or mathematical distributions, and gives some examples of fault rates for applications being executed on the Earth and also in space. Next, we present redundancy, which is a commonly used technique to provide systems with fault tolerance. Finally, the dynamic voltage and frequency scaling is described and we discuss whether its use is reasonable for systems aiming at maximising the reliability.

After this general context, the next two chapters focus on the *primary/backup approach*. While Chapter 2 presents the fundamentals, the related work and several applications, Chapter 3 covers our research. Its first part is devoted to independent tasks and the second one treats dependent tasks. For each type of tasks, we first introduce the task, system and fault models. Then, we describe our experimental framework and analyse the results. In particular, this chapter presents and compares different processor allocation policies and scheduling search techniques. It introduces the proposed enhancing techniques: the method

of *restricted scheduling windows*, the one of *limitation on the number of comparisons*, and the one of *several scheduling attempts*.

Chapters 4 and 5 deal with small satellites called *CubeSats*. Chapter 4 introduces and classifies them among other satellites according to their weight and size. We also mention the advent and show their progressive popularity and their missions. Next, we describe the space environment and how CubeSats are vulnerable to faults. Finally, we sum up the methods currently used to provide CubeSats with fault tolerance. To overcome the harsh space environment, Chapter 5 presents a solution to improve the CubeSat reliability. To analyse its performances, the system, task and fault models are defined and the three proposed scheduling algorithms are introduced. While the first two presented algorithms do not take energy constraints into account, the last devised algorithm is energy-aware. After the description of the experimental frameworks, the results in a fault-free and harsh environments are discussed.

Chapter 6 concludes the thesis by summing the main achievements and suggestions.

The thesis includes four appendices. Appendix A details how the exhaustive search of "boundary schedule search technique" was adapted for the "first found solution search: processor by processor", which does not carry out an exhaustive search. Appendix B lists and describes the input parameters when the directed acyclic graphs (DAGs) are generated using the task graph generator called DAGGEN. Appendix C presents several constraint programming parameters having influence on reproducibility of results and Appendix D explains the graphical representation of the box plot.

Chapter 1

Preliminaries

This chapter presents an overview of several topics closely related to the present manuscript of the PhD thesis. First, it sums up system, algorithm and task classifications. Second, it distinguishes terms associated with fault tolerant systems. Third, it summarises fault models and gives some examples of fault rates. Fourth, redundancy, which is one of the techniques to make system more robust against faults, is introduced. Fifth, the use of dynamic voltage and frequency scaling is discussed.

1.1 Algorithm and System Classifications

We present several types of classifications from the viewpoints of systems, algorithms and tasks. We remind the reader that the lists are not exhaustive and include terms, which allow us to clearly define our research problems in this manuscript.

We start to give two definitions. We call *mapping*, a placing of a task onto one of the system processors taking into account already scheduled tasks, and *scheduling*, a placing of a task onto one particular system processor taking into account already scheduled tasks on it.

To describe a system, its main characteristics related to scheduling are generally as reads:

— **Uniprocessor/Multiprocessor**
While a *uniprocessor* system has only one processor, a *multiprocessor* system has more than one. In general, scheduling on multiprocessor systems is a NP problem, which means that it is not easy to find an optimal solution and the use of heuristics is necessary. In fact, a problem is said to be NP, accounting for *nondeterministic polynomial time*, if it is solvable in polynomial time by a nondeterministic Turing machine. Such a machine is able to perform parallel computations without communications among them [151, 152].

— **Homogeneous/heterogeneous processors**
If a system is multiprocessor, it consists either of *homogeneous* or *heterogeneous* processors. Although systems composed of heterogeneous processors generally provide better performance because a scheduling algorithm can take advantage of distinct features of processors, the scheduling complexity is higher when compared to systems with homogeneous processors [132].
A more detailed classification formulated by Graham in [66] allows us to further characterise a system by conventional letters:
  — P denotes *identical parallel machines*, i.e. machines having the same processing frequency.
  — Q stands for *uniform parallel machines*, which means that each machine has its own frequency.
  — R represents *unrelated parallel machines*.
  — O means an *open shop*, i.e. each job J_j consists of a set of operations $O_{1j}, \ldots, O_{1m}$. The order of these operations is not important but O_{ij} has to be executed on machine M_j during p_{ij} time units.
  — F denotes a *flow shop*. Each job J_j is a set of operations $O_{1j}, \ldots, O_{1m}$ and the order of these operations has to be respected. O_{ij} has to be executed on machine M_j during p_{ij} time units.
  — J is a *job shop*. Each job J_j consists of a set of operations $O_{1j}, \ldots, O_{1m}$ and the order of these operations has to be respected. O_{ij} has to be executed on a given machine μ_{ij} during p_{ij} time units with $\mu_{i-1,j} \neq \mu_{ij}$ for $i = 2, \ldots, m$.

— **Real-time aspect**
Three categories are distinguished from the real-time point of view based on the respect of task

deadline [96, 117]. If *hard* real-time systems, such as space and aircraft applications or nuclear plant control, miss a task deadline, subsequent consequences may be catastrophic. For *firm* real-time systems, like online transaction processing and reservation systems, a respect of deadline is important because the results provided after the task deadline are not useful anymore but there are no dire consequences. Finally, the results delivered after the task deadline by *soft* real-time system, e.g. image processing applications, are utilisable but may be less pertinent.

A scheduling algorithm, which is generally run on a scheduler, has its main attributes as follows:

— **Online versus Offline**
An *offline*, also called *static* or *design-time*, algorithm knows all problem data in advance, for example number of tasks and their characteristics, such as arrival times, execution times or deadlines. When tasks arrive over time and a scheduling algorithm does not have any knowledge of any future tasks, the algorithm is called *online*, also named *dynamic* or *run-time* [117, 119, 125, 132, 134]. While online scheduling offers the possibility to adapt to system changes and task arrivals, it has higher computational cost than offline scheduling.
In case of online scheduling, we distinguish whether an algorithm is clairvoyant or non-clairvoyant [119]. While a *clairvoyant* algorithm is aware of all task attributes at the arrival time, a *non-clairvoyant* one notices that a new task arrives but the task characteristics are not available. For instance, the task execution time is known once a task was executed.

— **Competitive ratio**
To evaluate performances of an online algorithm a *competitive analysis* is carried out. An online algorithm A is called *c-competitive* if, for all inputs, the objective function value of a schedule computed by A is at most a factor of c away from that of an optimal schedule [125].
For example, we consider that we want to minimise an objective function of a given scheduling problem. For any input I, let $A(I)$ be the objective function value achieved by A on I and let $OPT(I)$ be the value of an optional solution for I. An online algorithm A is called *c-competitive* if there exists a constant b independent of the input such that, for all problem inputs I, $A(I) \leqslant c \cdot OPT(I) + b$ [125]. The competitive ratio is basically equivalent to a worst-case bound [119].

— **Global versus Partitioned**
If an algorithm schedules tasks on a multiprocessor system, there are two possibilities how it considers the system [117, 132]. If it considers only one task queue and one system sharing the resources, it is called *global* or *centralised*. Otherwise, each processor (or group of processors) has its own task queue and its own resources. In this case, we call it *partitioned* or *distributed* scheduling.

Regarding task characteristics, the ones related to our work are as follows:

— **Periodicity**
The periodicity defines whether a task is repeated or not. While a *periodic* task arrives at regular intervals, an *aperiodic* task arrives only once. To complete this classification, we mention that there are also *sporadic* tasks having a minimal time between two arrivals. Every task can be then further characterised, for example by arrival time, computation time, deadline or priority. When a new scheduling problem is introduced in this manuscript, a precise definition of task and its attributes are given (for more details see Sections 3.1.1 and 3.2.1 for primary/backup approach and Section 5.2.1 for CubeSats).

— **Precedence constraints**
Based on the existence of precedence constraints, we distinguish *independent* and *dependent* tasks.

— **Preemption**
A scheduling algorithm is called *preemptive* if it authorises to temporarily suspend running tasks having lower priority than a new arriving task and preferentially execute this new task. Otherwise, it is called *non-preemptive* and it does not interrupt currently executing tasks and new tasks can start after currently executing tasks finish their execution [117].

A scheduling problem is also defined by its optimality criteria. The most commonly used optimality

criteria are related to time performance, for instance completion time or lateness, but other objective functions become more and more frequent, such as energy consumption or reliability. Since demands on performance increases, one objective function may not be sufficient and a multi-objective problems are formulated. To solve such a problem, one can choose from three possibilities [125]:

— Transform some objectives into constraints.
— Decompose the multi-objective problem to several problems with a single objective, sort objective functions according to the importance and treat each objective separately.
— Use the Pareto curve to find an optimal solution, i.e. the one where it is not possible to decrease the value of one objective without increasing the value of the other [119].

Nowadays, systems are more and more vulnerable to faults and the reliability, i.e. the ability of a system to perform a required function under given conditions for a given time interval, becomes the main concern. Naithani et al. [105, 106] thus emphasized the necessity to consider the reliability aspect during scheduling. They showed that it is better to make use of reliability-aware scheduling rather than performance-optimised scheduling. On average, although the reliability-aware scheduling degrades performance by 6%, it improves the system reliability by 25.4% compared to the performance-optimised scheduling.

In order to standardise the classification of scheduling problems, Graham et al. [66] proposed a 3-field notation $\alpha|\beta|\gamma$ in 1979. The parameter α refers to the processor environment, while the parameter β represents task characteristics and the parameter γ stands for the objective function. An overview of scheduling algorithms based on Graham classification is available at the website `http://schedulingzoo.lip6.fr/`.

1.2 Fault, Error and Failure

If a system stops performing a required function, a chain of events occurs, as depicted in Figure 1.1. At the beginning, a source, such as a charged particle, activates a *fault*. This fault can then generate an *error*, which may propagate and cause a *failure*. Therefore, these three terms (fault, error and failure) are not the same and cannot be interchanged. Unfortunately, they are often confused and/or used interchangeably in literature. In this manuscript, we will stick to the terminology as defined above but we keep the original word when citing from different sources.

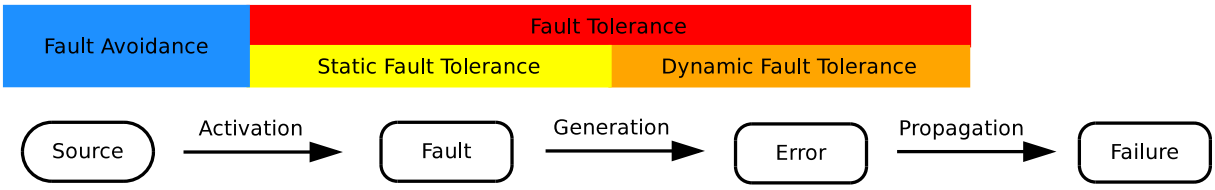


Figure 1.1 – Causal chain of failure (Adapted from [147, Figure 1.4])

Based on this terminology, we distinguish three terms when eliminating faults in the system. The *fault avoidance* tries to eliminate the activation of fault, whereas the *fault tolerance* aims to avoid its propagation to error (*static fault tolerance*) or to failure (*dynamic fault tolerance*).

Faults can have different origin and they can be classified in different classes. Several classifications were proposed by A. Avižienis et al. [16]. For example, they defined eight elementary fault classes, which are as follows:

— *Phase of creation or occurrence:* development and operational faults;
— *System boundaries:* internal and external faults;
— *Phenomenological cause:* natural and human-made faults;

— *Domain:* hardware and software faults;
— *Objective:* malicious and non-malicious faults;
— *Intent:* deliberate and non-deliberate faults;
— *Capability:* accidental and incompetence faults;
— *Persistence:* permanent and transient faults.

We note that one fault can be classified in several classes. For example, a charged particle in space can be classified as an operational, external, natural, non-malicious and non-deliberate fault. Its further classification then depends on the impact location (hardware or software) and duration (permanent or transient).

As regards the fault detection, isolation and recovery, there are various approaches and they are mainly application dependent. A general overview was presented in the previous work of the author [43, 44]. Consequently, since the primary/backup approach is a general method and the fault detection depends on its application, only list of general techniques is given in Section 3.1.1. Regarding CubeSats, the context is specific and, subsequently, a more detailed presentation of fault detection and recovery techniques is provided in Section 4.5.

1.3 Fault Models and Rates

We will introduce the processor failure rate and different possibilities how a fault injection and/or analysis can be carried out when evaluating algorithm performances. The first possibility is to make use of a Markov model, which is a probabilistic approach to evaluate the reliability of systems with constant failure rate. The second one is based on mathematical distributions, both discrete and continuous ones. Finally, different fault/failures rates in space and no-space applications will be compared.

1.3.1 Processor Failure Rate

Let us introduce the failure[1] rate λ, which is defined as the expected number of failures per time unit. In general, the failure rate varies in the course of time. There are more failures at the beginning of the lifetime due to not yet defined problems and at its end due to ageing effects. Therefore, its temporal representation depicted in Figure 1.2 resembles to a bathtub curve having three main phases: (1) infant mortality, (2) useful life and (3) wear-out.

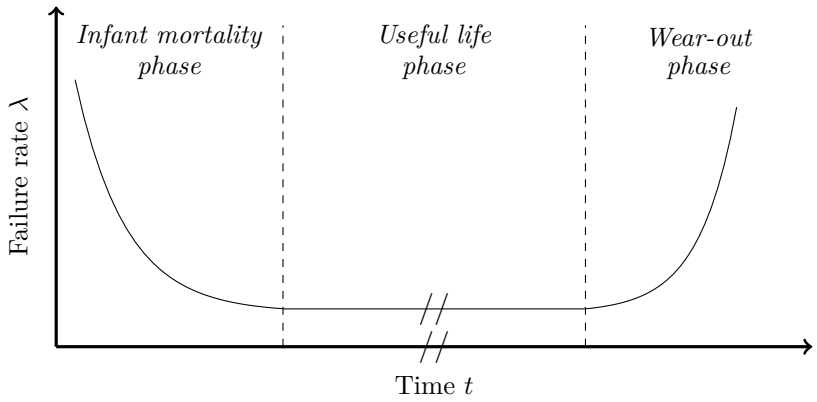


Figure 1.2 – Bathtub curve (Adapted from [85, Figure 2.1])

1. In literature, the terms *failure* and *fault* are often confused or used interchangeably. In this manuscript, while we keep the original word when citing from different sources, we stick to the terminology as defined in Section 1.2.

In general, the failure rate λ depends on many factors, such as age, technology or environment. In [85], the authors give the following empirical formula taking into account different factors:

$$\lambda = \pi_L \pi_Q (C_1 \pi_T \pi_V + C_2 \pi_E)$$

where
- π_L: learning factor, related to the level of technology development,
- π_Q: quality factor ($\in [0,25;20]$) accounting for manufacturing process quality control,
- π_T: temperature factor ($\in [0,1;1000]$),
- π_V: voltage stress factor for CMOS (Complementary Metal Oxide Semiconductor) depending on the supply voltage and the temperature ($\in [1;10]$), for other devices it is equal to 1,
- π_E: environment shock factor ($\in [0,4;13]$),
- C_1, C_2: complexity factors, functions of the number of gates on the chip and the number of pins in the package.

The failure rate can be also expressed as a function of the processor frequencies [88, 123, 148, 153]. We note $\lambda_{i,j}$ the failure rate of processor P_j when executing task t_i at frequency $f_{i,j}$. This rate can be computed as reads

$$\lambda_{i,j} = \lambda_i \cdot 10^{\frac{d(f_{max_i} - f_{i,j})}{f_{max_i} - f_{min_i}}} \tag{1.1}$$

where
- λ_i is the average failure rate of task t_i when the frequency is equal to the maximum frequency of task t_i denoted as f_{max_i},
- d is a constant indicating the sensitivity of failure rates to voltage and frequency scaling.

Formula 1.1 is frequently used to determine the value of λ when there are no reliability data for a studied system. Commonly used values for λ_i and d are summarised in Table 1.1.

Table 1.1 – Commonly used values of λ_i and d

Reference	λ_i	d
[158]	10^{-6}	$\{0, 2, 4, 6\}$
[41]	10^{-6}	3
[70, 71]	10^{-6}	4
[153]	$[2 \cdot 10^{-4}$ to $6 \cdot 10^{-4}]$	$\{2.1, 2.3, 2.5\}$
[88]	$[10^{-3}$ to $10^{-8}]$	$\{2, 3\}$

As an example of the system vulnerability, we mention that big cores are in general more vulnerable to bit flips than small cores because they consist of more transistors [105, 106]. Nevertheless, big cores execute faster, which reduces the exposure to faults during task execution.

Another possibility is to determine the failure rate using the probability theory [130] and exploit the reliability data that were already measured. It means that the value of λ or other parameters are computed based on a distribution. In order to determine parameters for a given distribution, once data of fault occurrences are available, they are analysed and modelled with different distributions (presented in Section 1.3.3) to find the best fit to the measured data.

1.3.2 Two State Discrete Markov Model of the Gilbert-Elliott Type

We introduce a Markov model, which is a probabilistic approach to evaluate the reliability of systems with constant failure rate [85].

The origin of name dates back to 1960, when E. N. Gilbert presented a Markov model of a burst-noise binary channel [62]. He considered two states: **G** (abbreviation of Good) and **B** (abbreviation of Bad

or Burst). In state **G**, transmission is error-free and, in state **B**, a digit is transmitted correctly with probability h. Three years later, E. O. Elliott improved this model and estimated error rates for codes on burst-noise channels [51]. At that time, the model was employed to provide close approximation to certain telephone circuits used for the transmission of binary data.

In 2013, M. Short and J. Proenza studied real-time computing and communication systems in a harsh environment, i.e. when a system is exposed to random errors and random bursts of errors [131]. They pointed out that, if the classical fault tolerant schedulability analysis is put into service, it may not correctly represent randomness or burst characteristics. Modern approaches could solve this issue but at the cost of increased complexity. Consequently, the authors decided to make use of the simple two-state discrete Markov model of the Gilbert-Elliott type to provide a reasonable fault analysis without significant increase in complexity. This model accounts for a "Markov-Modulated Poisson Binomial" process on one processor and it well represents errors, which are random and uncorrelated in nature but occur in short transient burst.

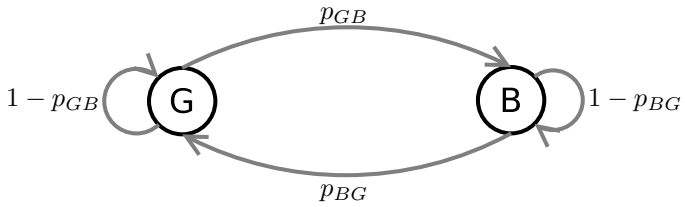


Figure 1.3 – Two state Gilbert-Elliott model for burst errors (Adapted from [131, Figure 1])

The two-state model is depicted in Figure 1.3. The probabilities to change the state are respectively p_{GB} and p_{BG} and the probabilities to remain in the same state are given as $p_{GG} = 1 - p_{GB}$ and $p_{BB} = 1 - p_{BG}$. The expected mean gap between error bursts is therefore defined as $\mu_{EG} = 1/p_{GB}$ and the expected mean duration of error bursts is determined by $\mu_{EB} = 1/p_{BG}$. The sum of μ_{EG} and μ_{EB} gives the expected interarrival time of error bursts. The probability of error arrival in each state is respectively defined as λ_B and λ_G. The reciprocal values of $1/\lambda_B$ and $1/\lambda_G$ denote the expected mean interarrival time of errors in a given state. The model parameters λ_B, λ_G, μ_{EG} and μ_{EB} are considered to have a geometric distribution, which is the discrete equivalent to the continuous exponential distribution.

The variable $m(t)$ is the probabilistic state of the Markov model at time t. It is encoded as the probability that the link is in state **B**, i.e. $m(t) = 1$. Therefore, the state at time $t + 1$ is computed using the following recurrent formula:

$$m(t+1) = p_{BB} \cdot m(t) + p_{GB} \cdot (1 - m(t)) = (1 - p_{BG}) \cdot m(t) + p_{GB} \cdot (1 - m(t)) \tag{1.2}$$

Regarding the initial condition, the authors considered the worst-case scenario, which means that the Markov chain starts in state **B**, i.e. $m(0) = 1$.

The probability that an error will arrive at time t is defined as:

$$p(t) = \lambda_B \cdot m(t) + \lambda_G \cdot (1 - m(t)) \tag{1.3}$$

In 2017, R. M. Pathan extended the previous model to multicore systems and add a new parameter related to the failure rate of permanent hardware faults [118]. He considers that multiple non-permanent faults can affect different cores at the same time, which means that Formulae 1.2 and 1.3 apply to each processor. Furthermore, he separates the error model and the fault model because the consequences of hardware faults, which manifest as errors on application level, depend on many factors, such as a fault detection mechanism or fault characteristics. For example, faults causing deadline misses are detected by watchdog timers and faults responsible for faulty output are identified by error-detection mechanisms.

In the case study of instrument control application, the values for five model parameters are as follows:

— Failure rate of permanent hardware faults in multicore chip: $\lambda_c = 10^{-5}/h$
— Failure rate of random non-permanent hardware faults in each core during a non-bursty period (**G** state): $\lambda_G = 10^{-4}/h$
— Failure rate of random non-permanent hardware faults in each core during a burst (**B** state): $\lambda_B = 10^{-2}/s$
— Expected duration of one non-bursty period (**G** state): $\mu_{EG} = 1/p_{GB} = 10^6\ ms$
— Expected duration of one bursty [2] period (**B** state): $\mu_{EB} = 1/p_{BG} = 10^2\ ms$

The Markov model is also used to compute the reliability in satellites, such as in [56]. In this publication, Markov models were analysed to study the reliability of on board computer (OBC) in four cases: a centralized OBC (which corresponds to the case represented in Figure 1.3), an OBC based on TMR, an OBC using task migration between two processors, and an OBC using task migration and three processors. It was shown that OBCs making use of task migration have higher reliability because when a processor is faulty, all tasks scheduled on it can be migrated to healthy processors. Therefore, the higher the number of processors, the higher the reliability.

1.3.3 Mathematical Distributions

The system reliability can be also modelled using several distributions. The most common distributions, which are used to model fault occurrences, are presented in this section after giving several definitions.

Let us consider a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, where
— Ω is a set of all possible outcomes,
— $\mathcal{F}$ is a set of events and a subset of Ω,
— $\mathbb{P}$ is a function assigning the probabilities to events [139].

We define a random variable X corresponding to a processor lifetime, i.e. the time until it fails. Since values of the lifetime are positive, X is mapped in real positive values, such as

$$\begin{aligned} X:\ \Omega &\longrightarrow \mathbb{R}^+ \\ \omega &\longmapsto X(\omega) \end{aligned}$$

We can then distinguish two cases: whether the random variable X is discrete or continuous.

1.3.3.1 Discrete Random Variable X

If X is a discrete random variable, then $X(\Omega)$ is a countable set, i.e. $X(\Omega) = \mathbb{N}$. An example of a discrete probability distribution is the ***Poisson distribution*** with parameter $\lambda > 0$ defined as:

$$\forall k \in \mathbb{N}, \mathbb{P}(X = k) = e^{-\lambda}\frac{\lambda^k}{k!} \tag{1.4}$$

where the value of k represents the number of faults.

The Poisson distribution assumes that faults are independent and the parameter λ denotes a constant failure rate per time unit. If this parameter is multiplied by time t (expressed in time units) to represent a specific number of occurrences within a given time interval [12, 162], then the probability that k faults occur in time t is given as:

$$\forall k \in \mathbb{N}, \mathbb{P}(X = k) = e^{-\lambda t}\frac{(\lambda t)^k}{k!}$$

and the reliability, i.e. the probability of zero failures in time t, is expressed as:

$$R = \mathbb{P}(X = 0) = e^{-\lambda t}\frac{(\lambda t)^0}{0!} = e^{-\lambda t} \tag{1.5}$$

2. Based on the related work, R. M. Pathan mentions that a burst length is $5\mu s$ [118].

The Poisson distribution is widely used in no-space applications [5, 41, 60, 68, 79, 88, 94, 99, 102, 123, 136, 143, 148, 153, 154, 156, 158, 159] because the assumption of constant failure rate is verified in most applications during the useful life, i.e. the second phase of the bathtub curve represented in Figure 1.2. Although this assumption may not be always valid in the harsh space environment, the Poisson distribution was also considered in space applications, for example in [32].

1.3.3.2 Continuous Random Variable X

If X is a continuous random variable, $X(\Omega)$ is in most cases an interval or a union of intervals and the probability is defined using a density function $f(t)$ [85, 139]. In our context, X is positive since it represents a processor lifetime and therefore the density function $f(t)$ satisfies:

$$\forall t \geqslant 0, f(t) \geqslant 0 \qquad \text{and} \qquad \int_0^{\infty} f(t)dt = 1$$

The cumulative distribution function of X, denoted by $F_X(t)$, represents the probability that the processor will fail at or before time t. It is defined as follows:

$$F_X(t) = \mathbb{P}(X \leqslant t) = \int_0^t f(\tau)d\tau$$

Finally, the reliability $R(t)$ is the probability that a processor will survive at least until time t, which means that

$$R(t) = \mathbb{P}(t < X) = 1 - F_X(t)$$

After giving the previous definitions, we now express the failure rate $\lambda(t)$, which is a conditional probability because we know that the processor correctly functioned at least until time t. It can be computed as reads:

$$\lambda(t) = \frac{f(t)}{1 - F_X(t)}$$

We note that the failure rate depends on a cumulative distribution function $F_X(t)$ and a probability density function $f(t)$. The probability density functions, which are often considered to model faults, are as follows [85, 91, 139]:

— ***Weibull distribution*** is defined as reads:

$$\begin{aligned} f(t) &= \lambda\beta t^{\beta-1} e^{-\lambda t^{\beta}} \\ \lambda(t) &= \lambda\beta t^{\beta-1} \\ R(t) &= e^{-\lambda t^{\beta}} \end{aligned}$$

where $\lambda > 0$ is a scale parameter and $\beta > 0$ is a shape parameter.
This distribution is appropriate to the bathtub curve, especially to model "infant mortality" and "wear-out" phases. Consequently, it can be considered as a general one to model well all cycle phases.
It is considered for example in [41, 133] for no-space applications. Regarding space applications, it is convenient to model all satellite lifetime. For example, researchers studied failure data from CubeSats and they found out that this distribution has the best fit to measured data [54, 92].

— ***Exponential distribution*** is characterised by

$$\begin{aligned} f(t) &= \lambda e^{-\lambda t} \\ \lambda(t) &= \lambda \\ R(t) &= e^{-\lambda t} \end{aligned} \tag{1.6}$$

The exponential distribution is a special case of the Weibull distribution with $\beta = 1$ and assumes a constant failure rate $\lambda > 0$, which is valid during the "useful life" phase of the bathtub curve. This distribution is commonly used due its simplicity.
It can be noticed [12] that, if we consider that a processor will not fail as long as no fault occurs, Formula 1.6, accounting for the reliability of exponential distribution, equals Formula 1.5, standing for the reliability of Poisson distribution.
Due to its simplicity, this distribution is commonly used to model faults in no-space applications, for example in [9, 25, 45, 70, 71, 82, 149], and even in space applications, for instance in [56].

— ***Lognormal distribution*** is expressed as follows:

$$f(t) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{\ln(x)-\mu}{\sigma}\right)^2}$$

where μ and σ denote the mean and standard deviation, respectively.
This distribution takes into account variations of failure rates throughout the processor lifetime [156] and it was used for example in [146] to model faults at the International Space Station.

In literature, when the reliability of task t_i is computed, the value of t generally represents the execution time of task t_i. For example, when considering the Poisson or exponential distributions, which have the same expression of the reliability (Formulae 1.5 and 1.6), the reliability of task t_i is

$$R = e^{-\lambda et_i}$$

where et_i is the execution time of task t_i, for example in [41, 70, 71, 88, 99, 123, 143, 148, 153, 158].

1.3.4 Comparison of Fault/Failures Rates in Space and No-Space Applications

The fault/failure rate also depends on the number of processors in the system. Intuitively, if there are more processors in the system, faults arrive more frequently.

In [72], the authors considered that a system consists of N identical processors and that each processor is characterised by its reliability measured by means of Mean Time Between Faults (MTBF) denoted by μ_{ind}. They proved that the overall system reliability μ is divided by the number of processors:

$$\mu = \frac{\mu_{ind}}{N} \tag{1.7}$$

which means that if we for example double the number of components, the system resiliency (in terms of the MTBF) is divided by two.

To complete the definitions, the MTBF is related to the Mean Time To Failure (MTTF) and the Mean Time To Repair (MTTR). If we consider that a processor repair is always perfect and the repaired system performs as the original system, these three times are related as follows [49]:

$$MTBF = MTTF + MTTR \tag{1.8}$$

In order to draw a comparison between fault/failures occurrences on the Earth and in space, fault/failure rates for no-space and space applications are respectively summed up in Tables 1.2 and 1.3. The data are classified according to fault/failure duration, i.e. whether only permanent or transient faults/failures or both together are considered. The rates are given per hour but in some cases they are expressed per time unit because the time unit was not specified in several papers. Last but not least, for each rate, we note a considered component (node, processor, system, ...).

The rates of permanent faults/failures are lower when compared to transient ones. In addition, rates are generally higher in space than on the ground because space is a harsh environment due to charged particles and radiation.

In the next three sections, we focus on three applications (high-performance computers, the International Space Station and CubeSats) to further analyse the fault/failure rates.

Table 1.2 – Fault or failure rates in no-space applications

`Permanent faults or failures`

Reference	Fault or failure rate	Application
[159]	*Node:* $1 \cdot 10^{-7}$ failure/hour	Heterogeneous clusters
[122]	*Processor:* $1 \cdot 10^{-6}$ to $7.5 \cdot 10^{-6}$ failure/hour	Heterogeneous systems

`Transient faults or failures`

Reference	Fault or failure rate	Application
[41, 158]	*Processor:* 10^{-6} fault/(unit of time)	Real-time embedded systems
[153]	*Processor:* $2 \cdot 10^{-4}$ to $6 \cdot 10^{-4}$ fault/(unit of time)	Heterogeneous embedded systems
[79]	*Node:* $1.5 \cdot 10^{-3}$ to $2.5 \cdot 10^{-3}$ failure/hour	Parallel system with 256 nodes
[158]	100 *megabit chip:* 10^{-5} to $\sim 10^{-3}$ fault/hour *Whole system:* 10^{-3} to $\sim 10^{1}$ fault/hour	Real-time embedded systems

`Transient and permanent faults or failures`

Reference	Fault or failure rate	Application
[99]	*System:* $1 \cdot 10^{-5}$ to $5 \cdot 10^{-5}$ failure/(unit of time)	Heterogeneous computing systems

`Type of faults or failures not mentioned`

Reference	Fault or failure rate	Application
[68]	*Processor:* $5 \cdot 10^{-6}$ to $15 \cdot 10^{-6}$ failure/hour	Heterogeneous computing systems
[143]	*Processor:* $1 \cdot 10^{-4}$ to $7.5 \cdot 10^{-4}$ failure/hour	Heterogeneous computing systems
[9]	*Processor:* $1 \cdot 10^{-4}$ to $1 \cdot 10^{-3}$ failure/hour	Homogeneous clusters

Table 1.3 – Fault or failure rates in space applications

`Permanent faults or failures`

Reference	Fault or failure rate	Application
[118]	*Processor:* 10^{-5} fault/hour	Safety-critical multicore systems

`Transient faults or failures`

Reference	Fault or failure rate	Application
[17, 32]	*System:* 10^{-2} to 10^{2} fault/hour	Different satellite and aircraft applications
[30]	512 *kbytes SRAM block:* 4 fault/hour	OBC of small satellite
[118]	*Processor:* 10^{-4} fault/hour (during no-bursty period) *Whole chip:* 10^{1} fault/hour (during bursty period)	Safety-critical multicore systems

`Transient and permanent faults or failures`

Reference	Fault or failure rate	Application
[56]	*Processor:* $1 \cdot 10^{-3}$ failure/hour	Satellite on-board computer (OBC)

1.3.4.1 Failures in High-Performance Computers

The High-Performance Computers (HPC) are computing systems consisting of several processors. Such systems are for example used for large-scale long-running 3D scientific simulations, such as plasma flow analysis [130]. The fault tolerance of HPC is frequently implemented as checkpointing [130]. This method consists in periodic saving of data during the execution. If a fault occurs in the course of task execution, it is restarted from the last checkpoint or from scratch if no checkpoint exists.

In this section, we consider the following HPC systems: Los Alamos National Laboratory (LANL),

Blue Waters, Tsubame, Mercury and one anonymous supercomputing site. The performances of these systems were already analysed in papers [15, 20, 130], which we will briefly summarise.

L. Bautista-Gomez et al. [20] evaluated occurrences of transient failures in Blue Waters, Tsubame, Mercury and LANL. They found out that there are periods with up to three times higher failure density when compared to other periods and they proposed dynamic checkpointing to detect such periods and save time.

If we assume that the MTTR is negligible when compared with MTTF and that the failure rate λ is constant, which is not accurate since the failure rate varies over system lifetime [130] (see Sections 1.3.1 and 1.3.3 for more details), we can approximately compute the number of failures per hour based on their data related to MTBF using Formula 1.8 and the following relation for failure rate:

$$\lambda = \frac{1}{MTTF}$$

Knowing the system characteristics, we then evaluate failure rates for studied systems and their cores thanks to Formula 1.7. The results are summarised in Table 1.4.

Table 1.4 – Failure rate of high-performance computers (Based on data from [20, Table 1])

System	MTBF (h)	Failures per h	# cores	MTBF (h)/core	Failures per h per core
Blue Waters	11.2	$8.93 \cdot 10^{-2}$	25 000	28 000	$3.57 \cdot 10^{-5}$
Tsubame	10.4	$9.62 \cdot 10^{-2}$	74 358	715 324	$1.40 \cdot 10^{-6}$
Mercury	16.0	$6.25 \cdot 10^{-2}$	891	14 256	$7.01 \cdot 10^{-5}$
LANL	23.0	$4.35 \cdot 10^{-2}$	24 101	327 888	$3.05 \cdot 10^{-6}$

It is also interesting to analyse origin of failures of four systems. As Figures 1.4 show, the majority of failures occurs in hardware and failures in software have the second largest percentage. The authors of [130], who analysed the failure data of LANL, stated that the most common hardware failure is due to central processing unit (CPU) (about 40%).

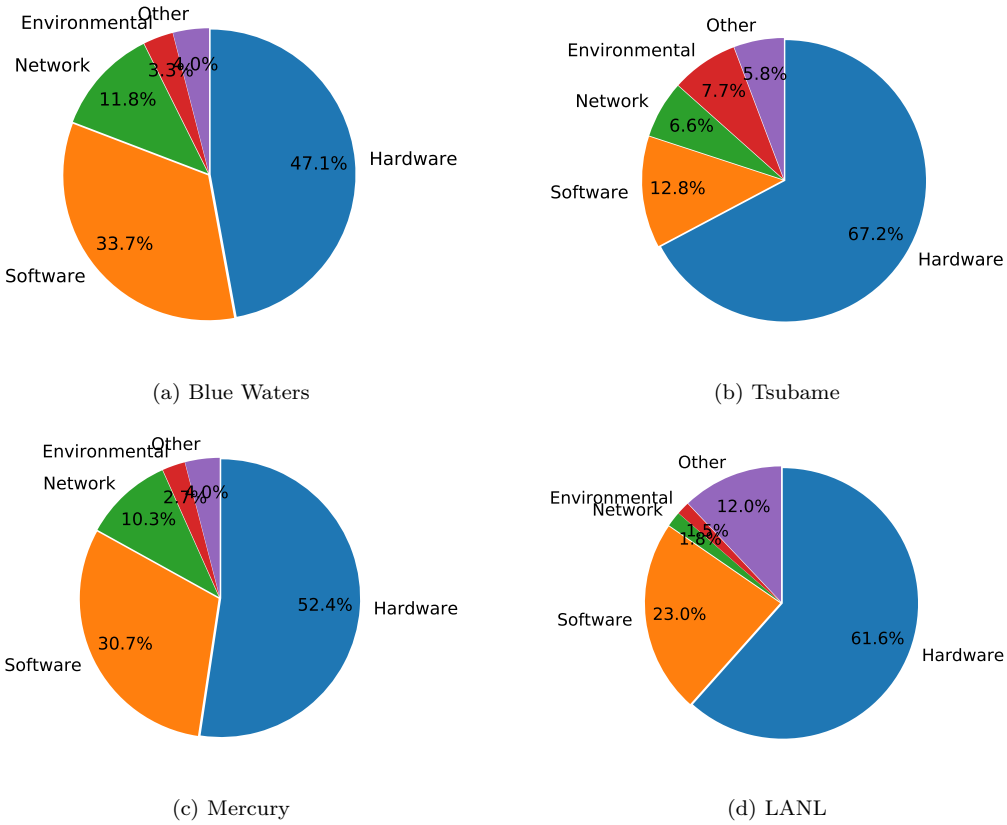
To give an example of failures, the authors of [20] found out that the most frequent failures of Mercury are as follows:

— errors in memory that were not correctable by Error Correction Code (ECC),
— processor cache errors,
— hardware-reported error in a device on the SCSI (Small Computer System Interface) bus,
— NFS (Network File System)-related error indicating unavailability of the network file system for a machine,
— PBS (Portable Batch System) daemon failure to communicate.

B. Schroeder and G. A. Gibson studied failure data from two HPC sites [130]. The first data set was collected over 9 years at LANL containing data from 22 high-performance computing systems (4750 machines and 24101 processors). The second data set was at an anonymous supercomputing site comprising 20 nodes and 10 240 processors. Their aim was to study statistical properties of the available data consisting of 23 000 failures.

First, they fitted their data using three probability distributions: the Poisson one, the normal one and the lognormal one. They estimated the maximum likelihood to parametrise the distributions and evaluated their fits. They found out that the Poisson distribution, which is often considered in fault analysis, does not represent well the data and that the normal and lognormal distributions fit better.

Moreover, when analysing the time between failures, the exponential distribution does not fit well but the gamma or Weibull distributions with decreasing hazard rate (Weibull shape parameter of 0.7-0.8) achieve better results. The *hazard rate* defines how the time since the last failure influences the expected time until the next failure. The studied data, which have a decreasing hazard rate, show that the longer the time since last failure, the longer expected time until next failure.



(a) Blue Waters

(b) Tsubame

(c) Mercury

(d) LANL

Figure 1.4 – Origin of system failures [20, Table 1]

Second, they stated that failure rates are approximately proportional to the number of processors, they fluctuate over a system lifetime and that they vary among systems, even for those having the same hardware type. Furthermore, failures are proportional to the workload and thus depend on the time of the day and on the day of the week. For instance, the results show that the failure rate is two times higher during peak hours than at night.

Third, the authors analysed space and time correlations between failures. They found out that there are no space correlations, i.e. one processor failure does not cause a failure of neighbouring processor, but time correlations, evaluated by autocorrelation, exist at all three time granularities (day, week, month). This means that the number of failures observed in one time interval is predictive of the number of failures expected in the following time intervals.

Since in many papers, the authors make use of assumption when studying HPC systems that faults are temporal independent, which is not correct as the preceding paragraph shows, G. Aupy et al. investigated further this assumption. They studied the failure logs from LANL and Tsubame using an algorithm to detect failure cascade based on the study of pairs of consecutive interarrival times [15].

On the one hand, they found out that it is wrong to assume failure independence everywhere but, on the other hand, they showed that the assumption of failure independence can be wrongly but safely used [15]. In fact, the knowledge of failure cascades, i.e. series of consecutive failures that strike closer in time than expected, does not bring a significant gain and the overhead of checkpointing due to assumption of failure independence is minimal.

1.3.4.2 Failure Rates at the International Space Station

R. Vitali and M. G. Lutomski analysed data from the International Space Station ISS, which is situated at the same orbit as CubeSats, to determine failure rates [146]. They studied different types of components divided in four categories: electronics (for example A/D converters), electrical (for instance Remote Power Control Module), mechanical (like pyro-valves) and electro-mechanical (such as electro-mechanical valves). They took into account Space Environment Conversion (SEC) factor [3] and they considered that all failure rates were independent in time and they can be modelled by a lognormal distribution, which was then confirmed by experiments.

The results, summarised in Table 1.5, show that the failures related to mechanics are more frequent than the ones related to electronics. Unfortunately, the paper [146] does not mention more details about systems, such as the number of processors. Consequently, we cannot make a detailed comparison between the failures rates of electronics at the ISS to the ones of other space applications presented in Table 1.3. Nevertheless, we note that the former values are lower.

Table 1.5 – Failure rates at the International Space Station [146, Table 1]

Category	Failure rate per hour
Electronics	$2.5 \cdot 10^{-6}$
Electrical	$3.0 \cdot 10^{-6}$
Mechanical	$2.5 \cdot 10^{-5}$
Electro-mechanical	$2.0 \cdot 10^{-5}$

1.3.4.3 Fault Injection in a CubeSat

Since data of fault rates in CubeSats are not easily available, we present how a fault injection on simulation level was carried out for a CubeSat.

It was realised by N. Chronas [37], who simulated faults by injecting errors in the Core Lock Step (CLS) design [4]. The faults were manually created by modifying the values at core outputs.

Since there is no model of the SEU generation in space, the timing for fault injection was determined by experiments using a test service equivalent to the "ping" request sent to network hosts. First, the author determined the period of test service at which the system starts losing packets without fault injection. He found 0.1 s. Then, he started to inject faults and the obtained results are summarised in Table 1.6. Finally, he said that the expected rate of faults induced by radiation is lower than the simulated rate. Actually, when the period of test service equals 0.1 s and a fault is injected every 0.05 s, the fault rate is $7.2 \cdot 10^4$ fault/hour.

1.4 Redundancy

A system is called *fault tolerant* if it continues to perform its specified function or service even in the presence of faults [49, 117].

To make system *fault tolerant*, i.e. more *robust* against faults, one of commonly used approaches is redundancy. *Redundancy* is the provision of functional capabilities that would not be necessary in a fault-free environment [49]. It can be in time or in space. *Time redundancy* consists in repeating the same computation or data transmission in order to make a comparison later and check for faults. *Space*

3. The *Space Environment Conversion* (SEC) *factor* converts the number of failures (k) during a specified time (t) that the component experiences in its native environment to the number of failures that would have been observed in space [146]. For instance, if there are 10 failures within a time interval t and SEC=2, the resulting adjusted number of failures during time interval t would be 5.

4. The *Lock Step technique* is a method to detect errors. Two cores execute the same code and their outputs are compared to detect a fault [138].

Table 1.6 – Fault injection into UPSat [37]

Period of test service (s)	**Fault injection average period (s)**	**Percentage of packet losses**
0.08	0	20
0.05	0	50
0.1	1	0
0.1	0.5	0
0.1	0.05	0
0.1	0.025	1.2
0.1	0.01	39.6
0.1	0.005	100

redundancy can be classified into three types depending on the type of redundant resources added to the system [44, 49, 85].

— *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 supplementary information, e.g. Reed-Solomon codes, Hamming codes, error-detecting parity codes or Cyclic Redundancy Check (CRC). This type of redundancy is mainly used to store or transmit data.

Although redundancy improves the system reliability, its overheads are not negligible.

For example, Goloubeva et al. [64] introduced additional executable assertions to check the correct execution of the program control for safety-critical applications. They showed that, depending on the application and program, the obtained overheads are considerable: memory ones (minimum: 124%, average: 283%, maximum: 630%) and performance ones (minimum: 107%, average: 185%, maximum: 426%).

To mention another example, Bernardi et al. [24] combined software-based techniques with an Infrastructure IP to detect transient faults in processor-based systems on a chip (SoCs). They tested several benchmarks and found out significant overheads in execution time (minimum: 78%, average: 126%, maximum: 209%), in code size (minimum: 68%, average: 162%, maximum: 270%) and in data size (minimum: 102%, average: 107%, maximum: 113%).

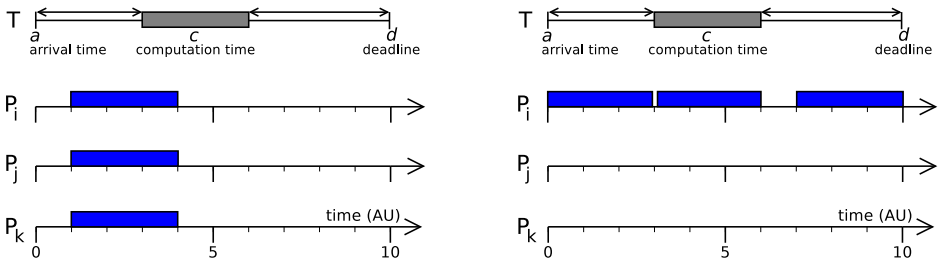
While it is not possible to prevent the overheads due to space redundancy, the ones caused by for time redundancy can be avoided. In fact, if after the first execution, no fault is detected, a new execution is subsequently not necessary.

Since the research carried out within the framework of the PhD thesis considers fault analysis at the task level, we give some definitions.

Redundancy at the task level considers that each task has one or several copies. If two identical copies of the same task are used, this approach is called *duplication* and it 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). Assuming that only one fault can occur at the same time, this technique is able to detect a faulty result and thanks to a majority voter chooses a correct one. *N-modular redundancy* (NMR) is a generalised version of TMR making use of N task copies.

Although replication of several task copies is considered in this thesis as one of the methods to improve the reliability, it can be also put into practice in other contexts. For example, task replication in [116] is used to reduce the schedule length by eliminating communication costs.

Both space and time redundancies can be used at the task level. The former one has an advantage not to delay the results in contrast to the latter one. Figures 1.5a and 1.5b respectively depict space and time redundancies for TMR.



(a) Redundancy in space (b) Redundancy in time

Figure 1.5 – Principle of redundancy

Whereas several authors consider that the number of task copies is fixed for a given algorithm, for instance two for the primary/backup approach [18, 61, 155], the others do not choose it in advance and let the algorithm make a decision based on the reliability [70, 71, 148]. An example of the latter case, published by Wang et al. [148], was implemented within the algorithm for *Replication-based scheduling for Maximizing System Reliability*. When scheduling a task, the algorithm dynamically computes the number of copies of a given task taking into account the processor reliability and the task reliability threshold γ, which is a parameter set by user. If the value of the present processor reliability is lower than γ, the algorithm determines the number of replicas, i.e. the number of task copies, to satisfy the reliability threshold γ. Otherwise, the replication is not required.

1.5 Dynamic Voltage and Frequency Scaling

This section introduces the Dynamic Voltage and Frequency Scaling (DVFS) and decides whether or not it is beneficial to systems aiming at high reliability.

The objective of DVFS is to decrease the voltage and/or to reduce a frequency when a processor executes a task in order to save energy. Although the use of DVFS is useful to optimise performance and power consumption, its control is more complexe [63].

The power consumption of the DVFS was evaluated by P. Duangmanee and P. Uthansakul in [48]. The relation between the power consumption i and the processor frequency f is as reads:

$$i = m \cdot f + i_{offset} \tag{1.9}$$

where m is a constant depending on the processor architecture and i_{offset} is the processor power consumption, which is independent on the frequency.

The processor energy consumed at voltage v and current i during the execution time t_{exec} is

$$E = v \cdot i \cdot t_{exec}$$

Since the execution time t_{exec} is related to the operating frequency as follows:

$$t_{exec} = \frac{k}{f}$$

where k is a constant depending on the processor architecture, we can express the processing energy at low frequency as reads:

$$E_{f_{low}} = v \cdot i\,(f_{low}) \cdot \frac{k}{f_{low}}$$

and the processing energy at high frequency in terms of low frequency as follows:

$$E_{f_{high}} = v \cdot i\,(m \cdot f_{low}) \cdot \frac{k}{m \cdot f_{low}}$$

The comparison of the energy consumed at high frequency $E_{f_{high}}$ to the one at low frequency $E_{f_{low}}$ gives:

$$\begin{aligned} E_{f_{high}} < E_{f_{low}} &\Leftrightarrow v \cdot i\left(m \cdot f_{low}\right) \cdot \frac{k}{m \cdot f_{low}} < v \cdot i\left(f_{low}\right) \cdot \frac{k}{f_{low}} \\ &\Leftrightarrow \frac{i\left(m \cdot f_{low}\right)}{m \cdot f_{low}} < \frac{i\left(f_{low}\right)}{f_{low}} \\ &\Leftrightarrow \frac{m \cdot f_{low} + i_{offset}}{m \cdot f_{low}} < \frac{f_{low} + i_{offset}}{f_{low}} \qquad \text{using Formula 1.9} \\ &\Leftrightarrow 1 + \frac{i_{offset}}{m \cdot f_{low}} < 1 + \frac{i_{offset}}{f_{low}} \end{aligned}$$

Since $\frac{i_{offset}}{m \cdot f_{low}} < \frac{i_{offset}}{f_{low}}$, the energy consumed at high frequency $E_{f_{high}}$ is lower than the one at low frequency $E_{f_{low}}$. It means that, at reduced frequency, the task execution is longer and the power consumption is higher due to static and frequency-independent energy [157].

From the point of view of the system reliability, when the frequency reduces and/or the voltage decreases, the occurrence of transient faults increases [70, 128, 153]. It was found [153, 157, 158] that it is difficult for an algorithm using the DVFS to optimise both reliability and power consumption at the same time. Actually, if the algorithm does not consider the reliability, the probability of failure is higher, whereas if it takes into account the reliability, the power consumption increases.

Finally, Xu et al. stated [153] that, when processors have high fault rates and any algorithm would hardly reduce the processor execution frequency, the algorithm without the DVFS generates the least energy consumption.

To conclude, the reliability and energy constraints act in the opposite manners because the improvement of one criterion degrades the other. All in all, the technique of the DVFS during task execution will not be considered in this manuscript.

1.6 Summary

This chapter presented several topics to introduce the reader to the context of the PhD thesis.

Firstly, system, algorithm and task classifications were given and, in particular, Graham's 3-field notation was presented. It will be used to classify our proposed algorithms.

Secondly, we clearly defined fault, error and failure.

Thirdly, we presented various tools to model faults and a processor failure rate. Fault rates for both space and no-space applications were compared.

Fourthly, we described redundancy, which is a commonly used technique to provide systems with fault tolerance.

And fifthly, we discussed the dynamic voltage and frequency scaling and concluded that we will not make use of this technique in this thesis.

Chapter 2

Primary/Backup Approach: Related Work

This chapter summarises the work already carried out on the primary/backup approach. First, it describes the advent of this approach and some already proposed enhancing techniques. Then, it presents several applications, where the primary/backup approach is successfully put into practice.

The terminology used in this section tries to use a reasonable trade-off between the terms originally published in papers and the terms employed in this thesis.

2.1 Advent

One of the first papers that suggested the use of a spare task copy in the case that a primary task copy fails was written by C. M. Krishna and K. G. Shin in 1986 [86]. The authors considered a dynamic programming algorithm for a multiprocessor real-time system dealing with tasks having hard deadlines. Since several processor failures may occur in this system, two types of copies of the same task are considered. These copies are named in their paper as a *primary clone* and a *ghost clone*. A ghost clone is activated if a processor fails and the corresponding primary clone or previously activated ghost clone cannot be correctly executed.

The aim of the proposed algorithm is to obtain an optimal schedule containing enough ghosts in order to sustain N_{sust} processor failures. The schedule is *locally-preemptive*, which means that clones placed on one processor can preempt other clones on the same processor but they cannot preempt clones on other processors. Consequently, the maximum number of preemption for a given processor is equal to the number of ghosts. In fact, if backup copies can be preempted by a primary copy in order to respect deadlines, the system achieves better schedulability than the baseline algorithm without preemption [145] but to the detriment of higher system complexity.

The presented algorithm is not straightforward and therefore we prefer to set the baseline algorithm for the primary/backup approach on the following works.

2.2 Baseline Algorithm with Backup Overloading and Backup Deallocation

The trilogy of papers [59, 61, 101] written by S. Ghosh, R. Melhem and D. Mosse in the 1990s laid the main foundations for the primary/backup (PB) approach. They proposed an algorithm meant for multiprocessor systems dealing with aperiodic real-time independent tasks. A task is characterised by its arrival time a, ready time r, deadline d and worst-case computation time c. A preemption is not authorised.

To provide the fault tolerance, each task has two copies: the primary copy and the backup one, which are scheduled on two different processors. Therefore, a system can tolerate at most one single fault of any processor at the same time because when a primary copy is impacted by a fault, the corresponding backup copy is executed. A fault can be transient or permanent but it is independent. It is detected using fail-signal processors, watchdogs, signatures or acceptance tests.

In general, primary copies are scheduled as early as possible and backup copies are placed as late as possible because primary copies are always executed and backup copies may not be necessary. Figure 2.1 depicts an example of scheduling of task T_i (with the assumption made by the authors that $a_i = r_i$). If a task cannot be scheduled between its arrival time and the deadline, i.e. there is not enough time to place its primary and backup copies, it is rejected.

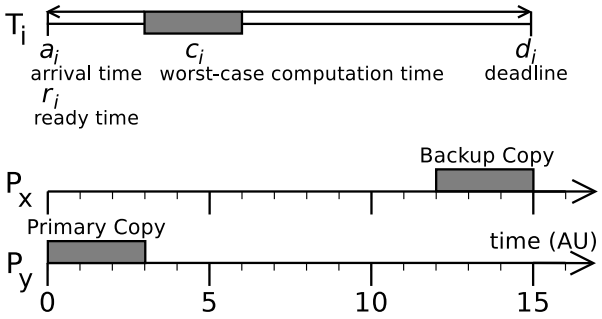


Figure 2.1 – Example of scheduling one task

In order to improve the schedulability and increase the processor utilisation two improving techniques were proposed: the backup overloading and backup deallocation. The *backup overloading* authorises several backup copies, if their respective primary copies are not scheduled on the same processor, to be overloaded, i.e. to share the same time slots on a processor, because only one backup copy will be necessary, if a fault occurs. An example of this technique is illustrated in Figure 2.2.

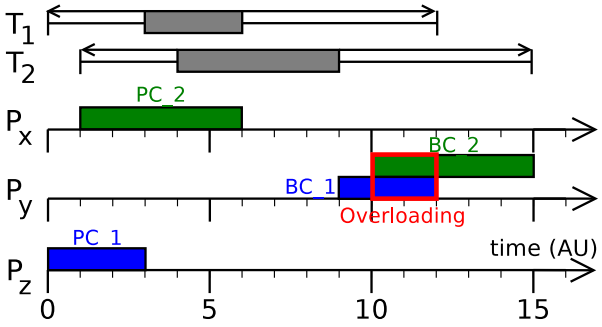


Figure 2.2 – Example of backup overloading

The *backup deallocation* means that a backup copy frees its slot once the corresponding primary copy is correctly executed.

The authors showed a dilemma of a scheduling choice for two aforementioned techniques. To favour the backup overloading, backup copies should be placed so that they overlap as much as possible, which is not necessarily as late as possible. To determine which technique is more effective, they chose the schedule maximizing the cost function Φ defined as follows:

$$\Phi = (\text{Start time of backup copy}) + \omega \cdot (\text{Overlap length}) \tag{2.1}$$

where the first addend is related to the backup deallocation and the second one is related to the backup overloading. The positive value of ω is fixed by user to set the prevalence of one of the aforementioned to another.

When introducing simple improvements for the primary/backup approach, we also mention that G. Manimaran and C. S. R. Murthy introduced *resource reclaiming* [96] in order to free slots when there are not necessary any more because a task copy finishes its execution earlier than originally scheduled.

To evaluate the system performances, the authors made use of two metrics. The *rejection ratio* accounting for the percentage of arriving tasks rejected by the system. The second metric is related to the system resiliency and called the *time to second fault*. It is the time it takes for the system to be able to tolerate a second fault after the first fault occurs.

The results show that; to reduce the rejection ratio, it is necessary to reduce the processor load, to add additional processors and/or to increase the window ratio (defined as $wr_i = \frac{d_i - r_i}{c_i}$). The time to second fault is longer when the number of processors decreases and/or the window ratio increases. When the primary/backup approach is compared to the system having a spare processor on which all backup copies are scheduled, its schedulability is higher due to better resource utilisation.

It was shown that both the backup deallocation and the backup overloading reduce the rejection ratio. Nevertheless, the backup deallocation performs better than the backup overloading because it is more advantageous freeing a slot on a processor than overloading already existing backup copies.

Finally, the authors noted that to handle multiple simultaneous faults it would be necessary to schedule more than one backup copies for each task, which will improve the system resiliency but to the detriment of overheads.

2.3 Processor Allocation Policy

When scheduling a new task and searching for a free slot, the choice of processor allocation policy plays an important role because it has a significant bearing on the system performances, such as the rejection rate or algorithm run-time. This section describes four types of searches: random, exhaustive, sequential and load-based.

2.3.1 Random Search

The *random search* randomly chooses one processor on which the algorithm tries to find a free slot [104]. A *free slot* is a time interval of processor schedule not occupied by any task copy and where a task copy of a new task can be placed. If a task copy is not placed on the first randomly chosen processor, a schedule of other randomly chosen processor is considered and so forth until the algorithm exhausts all possibilities or finds a free slot large enough to accommodate a task copy. The search is similar for both primary and backup copies.

2.3.2 Exhaustive Search

The *exhaustive search* was put into practice in the baseline approach presented in Section 2.2 [59, 61, 101]. This allocation policy tests all (P) processors to find a free slot as soon as possible for primary copy and ($P - 1$) processors to search for a free slot as late as possible for backup copy. Algorithm 1 sums up the main steps of the exhaustive search.

Algorithm 1 Algorithm using the exhaustive search

Input: Task T_i, Mapping and scheduling MS of already scheduled tasks
Output: Updated MS

```
1: if new task T_i arrives then
2:     for all (P) processors do
3:         Search for a free slot for primary copy
4:     if PC slot exists then
5:         Choose the slot situated as soon as possible
6:         for (P − 1) processors do
7:             Search for a free slot for backup copy
8:         if BC slot exists then
9:             Choose the slot situated as late as possible
10:            Commit the task T_i
11:        else
12:            Reject the task T_i
13:    else
14:        Reject the task T_i
```

On the one hand, this method is known to be the best for the primary/backup approach in terms of the rejection rate and processor load [155] because primary copies are scheduled as soon as possible and backup copies as late as possible. On the other hand, the algorithm needs to test all free slots within the scheduling window, which requires a non-negligible number of comparisons and therefore scheduling duration.

2.3.3 Sequential Search

When a system deals with hard real-time applications, it may not enough have time to search for a solution on all processors, assess all possibilities and then opt for the best one. Therefore, it is essential to devise a policy which can quickly provide a solution. Naedele [103] suggested the *sequential search*.

The algorithm using this processor allocation policy goes through processors, one by one, until it finds a free slot large enough to place a task copy or until it scours all processors. Inasmuch as all possibilities are not tested, the found solution may not correspond to the best one.

In order to avoid non-uniformity of processor load, the sequential search for primary copy starts on the processor following the processor on which the primary copy of previous task was successfully scheduled. The search then continues in increasing order until a free slot is found or no more processor is available [104]. If the primary copy of a new task is found on processor P_i, a search for a free slot for backup copy is carried out. It starts on processor P_{i-1} and it continues in decreasing order of the processors till a free slot is found or no more processor is available.

Algorithm 2 summarises the main steps of the algorithm based on the sequential search.

Algorithm 2 Algorithm using the sequential search

Input: Task T_i, Mapping and scheduling MS of already scheduled tasks
Output: Updated MS

```
1: if new task T_i arrives then
2:     while PC_i slot not found do
3:         Search for a free slot for primary copy
4:     while BC_i slot not found do
5:         Search for a free slot for backup copy
6:     if PC and BC slots exist then
7:         Commit the task T_i
8:     else
9:         Reject the task T_i
```

2.3.4 Load-based Search

Naedele [104] presented also another processor allocation policy. It is based on processor load. Before searching for a free slot, the algorithm evaluates the current processor load and it orders processors in a list according to their increasing workload. The search then starts on the least loaded processor. The search for a free slot for primary copy is carried out on odd processors and the one for backup copies is conducted on even processors until a solution is found or all possibilities are tested.

This method seems to be well applicable for dynamic mapping and scheduling [18] but its implementation may require more resources and algorithm run-time than the previously mentioned processor allocation policies. When these different policies were compared, it was found out that the sequential search and the load-based one have similar performances [104].

The main scheduling steps of the algorithm using the load-based search are shown in Algorithm 3.

Algorithm 3 Algorithm using the load-based search

Input: Task T_i, Mapping and scheduling MS of already scheduled tasks
Output: Updated MS

```
1: if new task T_i arrives then
2:     Order processors by their increasing load
3:     while PC_i slot not found do
4:         Search for a free slot for primary copy on odd processors
5:     while BC_i slot not found do
6:         Search for a free slot for backup copy on even processors
7:     if PC and BC slots exist then
8:         Commit the task T_i
9:     else
10:        Reject the task T_i
```

To sum up, the random processor allocation is not ingenious and the exhaustive search is more likely to be complex. The load-based and sequential processor allocation policies achieve good results but the former one is more complex due to system monitoring.

2.4 Improvements

We present improvements already proposed for the primary/backup approach. After the description of the primary slack and decision deadline, the difference between the passive and active primary/backup approach is explained. Finally, the computation of replication cost is introduced, as well as the boundary schedules and the primary-backup overloading.

2.4.1 Primary Slack

A condition may sometime occur that a free slot is not large enough to accommodate a task copy $xC_{to\_be\_scheduled}$. The solution proposed by [61] is to move forward already scheduled primary copies (without violating their time constraints) in order to increase the length of the free slot if it is not large enough to place a copy of a new task. That is why, the primary copy $PC_{to\_be\_moved\_forward}$, which hinders scheduling a task copy $xC_{to\_be\_scheduled}$, can be moved forward if there is another free slot after $xC_{to\_be\_moved\_forward}$ and if both tasks $T_{to\_be\_moved\_forward}$ and $T_{to\_be\_scheduled}$ respect their respective deadlines. Ghosh et al. defined the *slack*, as the maximum time by which the start of a task can be delayed to meet its deadline.

The backup copies are not concerned by this technique because they are scheduled as late as possible and they consequently cannot be moved forward. Moreover, several backup copies can be overloaded, which complicates the use of slack for backup copies.

To illustrate the primary slack, we consider a 2-processor system with two already scheduled tasks T_1 and T_2. At $time = 3$, a task T_3 arrives, as shown in Figure 2.3. The algorithm searches for a free slot and it finds out that there are no free slots for both primary and backup copies using the baseline algorithm. Since the algorithm makes use of primary slack, it realises that, if the primary copy PC_1 is moved forward, the primary copy PC_3 can be placed before PC_1 on processor P_1. The backup copy BC_3 is mapped on processor P_2.

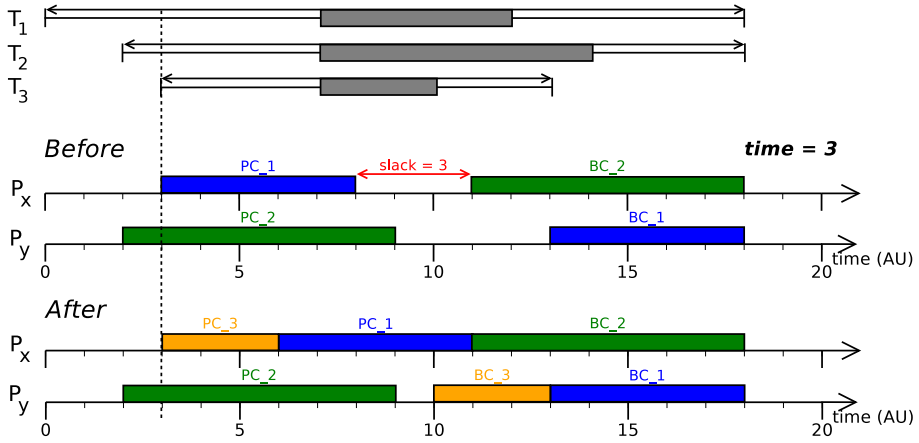


Figure 2.3 – Example of the primary slack (Adapted from [43, Figure 2.7])

Although the principle of this method was presented in [61], its evaluation was carried out in [104] by Naedele. He found that the unlimited use of slack to schedule a task only marginally contributes to decrease the rejection rate. Thus, he suggested to implement only one shift of already scheduled task. The results show that the benefit of the primary slack depends on the task set. On the one hand, if it is composed of tasks with a lot of slack, this method facilitates the reduction of the rejection rate. On the other hand, this technique can perform worse than the baseline primary/backup approach algorithm.

2.4.2 Decision Deadline

In the preceding section, a slack was used to extend a slot in order to place a task copy. Another method to improve schedulability is based on postponing the decision whether a task is accepted or rejected. In the baseline primary/backup approach, this decision is made at the task arrival. Nevertheless, if a new scheduling attempt took place later, one or several backup copies could be deallocated and consequently there could be enough space to schedule T_i within its deadline.

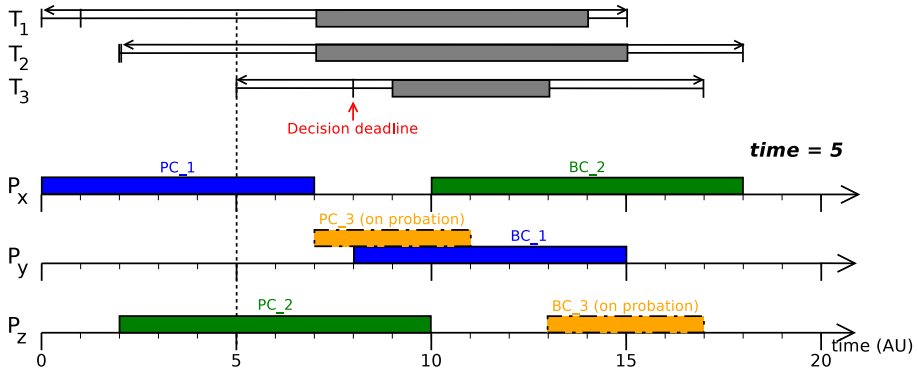
An improvement based on the postponement of the decision was proposed by Naedele [103, 104]. He considers that every task has one additional characteristic, which is called the *decision deadline* and denoted d_d. Therefore, if a task T_i is not accepted at the task arrival time but the algorithm finds that the copy after the found free slot is a backup copy $BC_{already\_scheduled}$ and the corresponding primary copy $PC_{already\_scheduled}$ finishes before d_d, the task is scheduled on probation. In such a case, when the primary copy $PC_{already\_scheduled}$ finishes its execution, a fault detection mechanism informs whether a fault occurred and, if negative, the backup copy $BC_{already\_scheduled}$ is deallocated and the copy of the task T_i can be definitively scheduled, if the current free slot is sufficient. This method is applicable to schedule both primary and backup copies.

As an example, we consider a 3-processor system, where two tasks have been already scheduled, as depicted in Figure 2.4. The second vertical line was added to the task model to refer to the decision deadline. The decision deadline can be equal to the task arrival time (task T_2) or later (tasks T_1 and T_3).

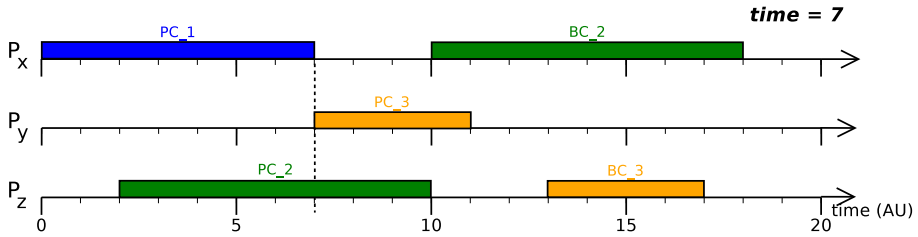
As it is illustrated in Figure 2.4a, the task T_3 arrives at $time = 5$ and the algorithm tries to schedule it. Nevertheless, there is no free slot large enough to schedule the primary copy PC_3. Consequently, the

algorithm schedules the task on probation and waits until the end of the execution of the primary copy PC_1 to decide whether the backup copy BC_2 can be deallocated.

At $time = 7$, the fault detection mechanism does not report any fault, the backup copy BC_2 is deallocated and the task T_3 is definitively accepted on the system as shown in Figure 2.4b.



(a) Task T_3 is scheduled on probation



(b) Task T_3 is definitively scheduled

Figure 2.4 – Example of the decision deadline (Adapted from [43, Figure 2.5])

2.4.3 Active Approach

The primary/backup approach schedules two task copies. In general, primary and backup copies of the same task do not overlap in time on two different processors. It means that there is enough time between the arrival time and deadline, i.e. the task window is two times larger than the computation time. This approach is called the *passive approach*. As this approach is not suitable for task with tight deadlines, Tsuchiya et al. [144] suggested to authorise the backup copy to overlap the corresponding primary copy on two different processors. This approach is called the *active approach* and an example of this technique is illustrated in Figure 2.5.

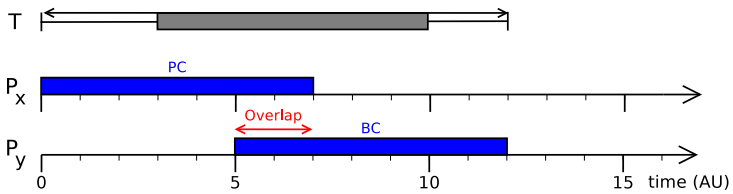


Figure 2.5 – Principle of the active primary/backup approach (Adapted from [43, Figure 2.4])

Although the active approach is well adapted to schedule tasks with tight deadlines, which is important for real-time systems, its drawback consists in giving rise to system overheads. In fact, when both primary

and backup copies of the same task are overlapping, the system entirely or partially performs the same computations twice.

To reduce overheads, it was suggested [144] that, if a primary copy correctly finishes earlier its execution than the corresponding backup copy (since the backup copy started its execution later), the remaining part of the backup copy can be deallocated. This modification slightly reduces the rejection rate.

Taking into account the system overheads, when the active approach is put into practice, it is only used in the case a task cannot be scheduled using the passive approach, such as in [4, 26, 159].

2.4.4 Replication Cost and Boundary Schedules

In [155], while primary copies are always placed as soon as possible (after an exhaustive search on all processors), scheduling of backup copies is not so straightforward because their position plays an important role in the system schedulability, in particular when a system deals with dependent tasks (see Section 2.6).

To solve the problem, the authors mainly focus on scheduling of the backup copies and they present two ideas: (i) the *replication cost* when scheduling the backup copies with the backup overloading, and (ii) the *boundary schedules* to reduce number of tests during search for a slot to place a task copy.

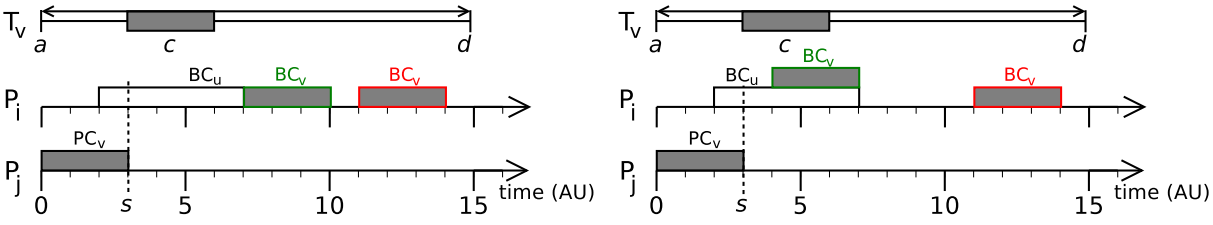
The *replication cost* is the percentage of time during which a backup copy is not overlapping with any other already scheduled backup copies to its computation time. It is defined as follows:

$$\text{Replication cost} = \frac{(\text{computation time of backup copy}) - (\text{duration of overloading})}{\text{computation time of backup copy}}$$

As an example, if a backup copy fully overloads other backup copies, its replication cost is 0. The replication cost is evaluated for each possible slot for the backup copy.

In order not to evaluate it for all slots on each processor and thus avoid the high complexity, they consider only "boundary schedules" of the backup copies. A *boundary schedule* is a slot having its start time and/or finish time at the same time as the beginning or end of an already scheduled task copy. Therefore, we note that the term "schedule" is not properly used in [155] for it has rather a meaning of "slot". In fact, the aim of their techniques is to deal with allotted places (=slots) within an arrangement (=schedule) and not to make any changes of already scheduled tasks. The aim is to make use of overloading as much as possible and do not test all possibilities. Actually, the authors showed that "boundary schedules" always have lower or the same replication cost but earlier completion time than slots which are not boundary.

Figures 2.6 depict an example of boundary (green) and non-boundary (red) "schedules". The green backup copy BC_v illustrated in Figure 2.6a does not overlap with the backup copy BC_u and thus its replication cost is 100%, while the one represented in Figure 2.6b has its replication cost 0%.



(a) Replication cost of the green backup copy $BC_v = 100\%$ (b) Replication cost of the green backup copy $BC_v = 0\%$

Figure 2.6 – Example of boundary (green) and non-boundary (red) "schedules"

Using aforementioned techniques, the authors devised the algorithm called *Minimum Replication Cost with Early Completion Time* (MRC-ECT). It is designated for independent tasks and aims at improving resource utilisation by minimising replication cost and therefore favouring the backup overloading. In case of a tie, a slot with the earliest completion time is chosen.

On the one hand, the results show that the proposed algorithm rejects less tasks and has lower replication cost than the algorithm scheduling all backup copies as soon as possible. On the other hand, the response time of the backup copies, i.e. the time when results are available, is longer.

Inspired by these enhancements, J. Balasangameshwara and N. Raju [18] proposed a fault tolerant load-balancing algorithm aiming at reducing replication cost and completion time. The algorithm is dynamic, adaptive and decentralised, which means that all system resources contribute to balance the system load. The *resource load* at a given time instant is defined as the total length of the jobs in the queue, where tasks are waiting to be executed, divided by the current capacity of the resource.

The authors consider a heterogeneous system dealing with independent tasks only. The devised algorithm takes into account communication costs among resources, such as transfer delay and data transmission rate, and can tolerate transient and permanent faults assumed to be independent. Regarding the fault model, the authors consider a fault-detection mechanism to detect faults and that only one version of a job, i.e. one job copy, can encounter a fault.

In order to evaluate the resource load, the authors propose a *resource efficiency estimation policy*, which means that each submitted but not yet executed job adds one point to the resource score and when it is executed one point is deducted. The lower the resource score, the higher the resource efficiency.

The goal of the load adjustment policy is to reduce the difference in load among resources by task migration. A task can migrate if it is waiting in a queue and other resource is less charged. To limit the task migration and avoid burdening several resources, the authors define a threshold for the maximum exchange in the system.

Other improvement proposed in this paper is the refinement of the *mutual information feedback* by adding information concerning the efficiency. Thus, each resource is aware of state (load and efficiency) of its neighbouring resources. These data are approximate because, in order to reduce communication costs, there is no message exchange unless there is a task transfer request between resources to which these data are appended.

The proposed algorithm guarantees to find an optimal backup slot, which reduces replication cost and thus contributes to higher utilisation efficiency but at the cost of testing all resources (1000 in their experiments). It was shown that a resource should exchange state message with only several neighbouring resources in order not to increase the response time. Furthermore, when the load increases, the average response time increases and the replication cost remains stable. Finally, the system fault tolerance is independent of the system heterogeneity and of the number of jobs in one job set.

2.4.5 Primary-Backup Overloading

In many papers, the authors assume that there is only one fault in the system at the same time. Consequently, when using the primary/backup approach with the backup overloading, several backup copies can overlap each other because only one copy will be executed in case of a fault occurrence.

R. Al-Omari et al., inspired by the backup overloading, proposed the primary-backup overloading to improve the schedulability in the multiprocessor real-time systems [5]. This technique requires the backup deallocation and authorises a primary copy of one task to overlap a backup copy of another task. Therefore, it is necessary to distinguish two states of the backup copies based on whether or not a backup copy takes part in the primary-backup overloading and subsequently cannot be overloaded anymore. The changes of states of the backup copies are encapsulated in Algorithm 4 and they are respectively denoted $PB\_overload\_authorised$ and $PB\_overload\_forbidden$ [43].

To illustrate the primary-backup overloading, an example is depicted in Figure 2.7a. At $time = 6$, we consider a 3-processor system having three already scheduled tasks, when a task T_4 arrives. Its primary copy is scheduled on the processor P_1 where it overlaps with the backup copy of the task T_1. We note that it would not be possible to schedule the task T_4 without the primary-backup overloading. At $time = 7$, there are two possibilities:

1. The primary copy PC_1 is correctly executed and the backup copy BC_1 is deallocated. Thus, the primary copy PC_4 can continue its execution. This scenario is illustrated in Figure 2.7b.

Algorithm 4 Implementation of the primary-backup overloading

```
1: if primary copy PC_i has just been scheduled then
2:     if PC_i is scheduled without overloading with another backup BC_j then
3:         Update the state of BC_i to PB_overload_authorised
4:     else
5:         Update the state of BC_i to PB_overload_forbidden

6: if primary copy PC_k has just finished then
7:     if BC_k overlaps with PC_i then
8:         if no fault occurs during PC_k then                    ▷ BC_k is deallocated
9:             Update the state of BC_i to PB_overload_authorised
10:        else                                                   ▷ BC_k cannot be deallocated
11:            Update the state of BC_k to PB_overload_forbidden
```

2. A fault occurs during the execution of the primary copy PC_1 and the system waits for the results of the backup copy BC_1, as shown in Figure 2.7c. Subsequently, the primary copy PC_4 cannot be executed and the result of the task T_4 will be known after the execution of the backup copy BC_4.

As this example shows, the primary-backup overloading increases the schedulability but at the cost of higher time to second fault. In [5], the authors evaluating this technique concluded that the schedulability is better about 25% compared to the backup overloading and that the upper bound of the time to second fault is twice as high as the time to second fault for the backup overloading.

W. Sun et al. [142] then put together the backup overloading and the primary-backup overloading and called it the *hybrid overloading*. It means that two overloading techniques jointly work to improve the system schedulability. It was shown that the hybrid overloading achieves an acceptance ratio similar to the one of the primary-backup overloading and that the value of the time to second fault is between the one for the primary-backup overloading and the one for the backup overloading.

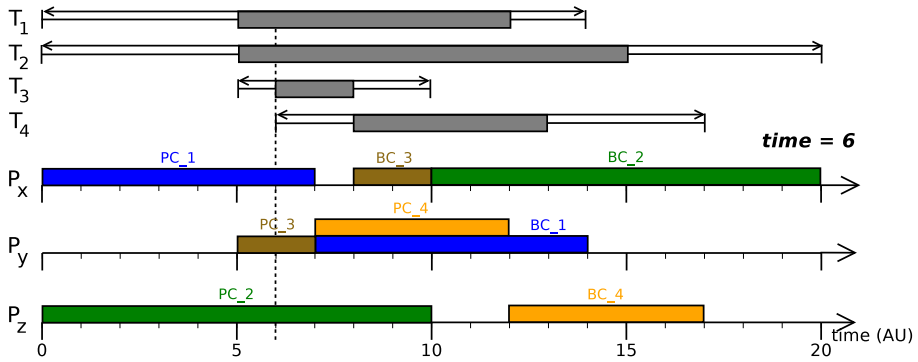
2.5 Fault Tolerance of the Primary/Backup Approach

Although a great deal of research has been conducted on the scheduling algorithms for the primary/backup approach, only few studies evaluating their resiliency have been published, despite the fact that this topic is of major concern for the embedded systems. In this section, we sum up several approaches.

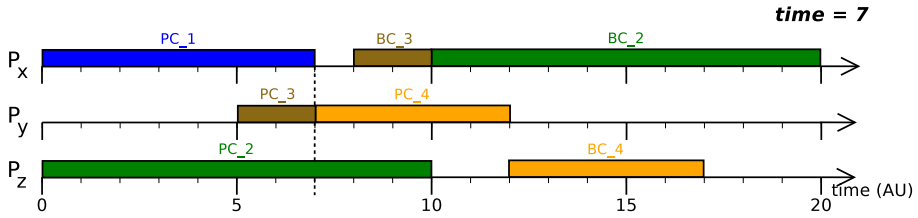
At the beginning, no faults were injected and the system resiliency were evaluated by means of a metric called the *time to second fault* [61]. It is the time it takes for the system to be able to tolerate a second fault after the first fault occurs. S. Ghosh et al. [61] considered only one transient or permanent independent fault in the system at the same time and they showed that this time is longer when the number of processors decreases and/or the window ratio increases.

G. Manimaran and C. S. R. Murthy [96] associated each primary copy with the probability that this copy fails. The values were between 10% and 50%. They assumed that there may be more than one transient or permanent independent faults all at once in the system because processors are divided into groups and each group can tolerate one fault. They found that the higher the probability that the primary copy fails, the lower the guarantee ratio.

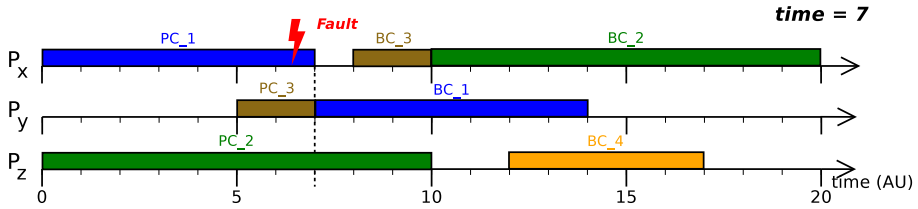
H. Kim et al. [83] disproved a hypothesis made in [60], where the authors assumed that there is at most one fault within time interval Δf since this assumption is not always valid as shown in Figure 2.8a. Consequently, they stated that it is necessary to consider inter-fault time ΔF, which is the time between one fault that occurred and the next fault. It is depicted in Figure 2.8b. They still consider that faults are transient or intermittent and that the system can tolerate only one fault. In their simulation, faults were generated with the fault rate 0.2 so that the minimum fault time $\Delta F = 200$ (time units). The results showed that the presence of faults greatly affects the rejection rate. They conducted 10 simulations and the rejection rate varied from several percents up to 20%.



(a) Primary-backup overloading of tasks T_1 and T_3

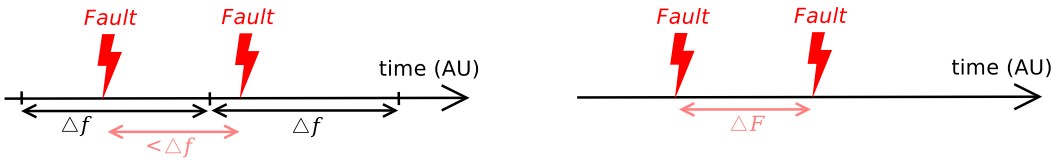


(b) BC_1 can be deallocated



(c) BC_1 cannot be deallocated

Figure 2.7 – Example of the primary-backup overloading (Adapted from [43, Figure 2.6])



(a) Fault interval Δf

(b) Minimum inter-fault time ΔF

Figure 2.8 – Difference between Δf and ΔF (Adapted from [83, Figures 1 and 2])

Similarly to the previous work, H. Beitollahi et al. [21] injected faults based on a value of the mean time to failure (MTTF). Since this research considered only a uniprocessor system, a transient fault was considered. They concluded that the larger MTTF, i.e. the lower the failure rate, the lower the number of lost tasks.

The authors of papers [9, 122] considered only one processor permanent failure and modelled the system reliability as follows:

$$R = e^{-\lambda c}$$

where λ is a processor failure rate and c is a task computation time. As expected, they found out that the higher the fault rate, the lower the reliability.

Since faults are in general a random phenomenon in nature, R. Sridharan and R. Mahapatra [136] suggested to make use of a stochastic process to model transient faults. Thus, faults were generated using the Poisson distribution and injected at task level. The authors were interested in the response time, which is directly related to the energy consumption. The higher the number of faults injected, the higher the energy consumption due to increased response time.

X. Zhu et al. [159] proposed a QoS-aware (quality of service) fault tolerant scheduling algorithm dealing with transient and permanent independent faults. At the same time, there is at most one fault. Faults were uniformly distributed with the fault rate of node equal to $10^{-7}/h$ and it was shown that the reliability cost computed for the system is almost independent of the number of nodes, task arrival time, task deadline, task heterogeneity and system heterogeneity.

2.6 Dependent Tasks

In general, dependent tasks are modelled by the *directed acyclic graph* (DAG) $G = \{V, E\}$ where V stands for a set of non-preemptable tasks and E denotes a set of directed edges representing communication among tasks. Every DAG is characterised by arrival time, deadline and computation time for each task. An example is depicted in Figure 2.9.

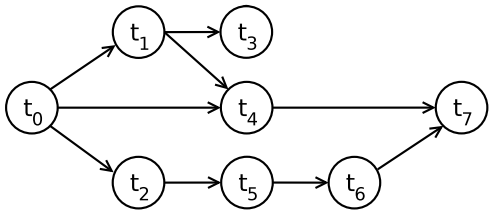


Figure 2.9 – An example of the general directed acyclic graph (DAG)

X. Qin and H. Jiang [121] present the *efficient fault tolerant reliability-driven algorithm* (eFRD), which is an offline scheduling algorithm based on the primary/backup approach. It can deal with hard real-time non-preemptable tasks with precedence constraints. This algorithm is reliability-aware because tasks are allocated to processors having high reliability.

The system consists of P heterogeneous processors with fully connected network. The computational heterogeneity is represented by different task execution times on each processor in the system. It can tolerate one processor permanent fault. The authors assume that the fault arrival rate is constant and the distribution of the fault-count for any fixed time interval is approximated using the Poisson probability distribution.

The algorithm schedules the primary and backup copies as soon as possible. Both copies of the same task can overlap neither in time nor in space. First, it searches for a primary copy slot on all processors and it chooses the slot having the highest reliability (computed based on processor failure rates) and, in case of a tie, the earliest slot is selected. Second, it looks for a backup copy slot on all processor, except the one accommodating the corresponding primary copy, and it opts for the slot having the highest reliability and the earliest start time.

Regarding the proposed algorithm eFRD, the results show that the more precedence constraints among tasks, the more messages exchanged and consequently the less task parallelism available, which has an impact on the system schedulability and reliability. Furthermore, the authors carried out simulations using 9 through 16 processors and they stated that the reliability and schedulability remain constant regardless of the number of processors.

In Section 2.4.4, we have already mentioned the work published in [155] regarding the replication cost and boundary schedules. Q. Zheng et al. focus on online scheduling of not only independent tasks but

also dependent ones. Scheduling dependent tasks is more complicated because there are dependencies among tasks, which constraint their scheduling.

While the primary copies can start as earlier as possible, once the results of predecessors are available, the backup copies need to respect time and space constraints. They cannot be scheduled on a processor where a primary copy of one of dependent tasks is mapped and they can start once results from their predecessors are available. Moreover, the backup overloading is constrained as well. All in all, the scheduling of the backup copies of dependent tasks is not straightforward.

To make such a computation easier, the authors proposed the algorithm to determine the earliest possible start time of a backup copy when scheduling dependent tasks. In order to improve the schedulability and loosen several space constraints, they consider the maximum fault recovery time. This means that even if a fault occurs, a task copy is recovered within this time. Therefore, all scheduling constraints for backup copies outside of this time interval are not considered. This refinement is especially useful for scheduling of large-scale dependent jobs and was already used in previous research, such as in [83].

The authors designed the algorithm called *Minimum Completion Time with Less Replication Cost* (MCT-LRC). It is meant for dependent tasks and tries to reduce the rejection rate by minimising completion time of each backup copy. In case of a tie, a schedule with less replication cost is chosen. Regarding that this algorithm takes into account all dependencies (comparing to [121] where only one direct predecessor is considered), the authors sum up conditions to preserve fault tolerance, like time and spatial constraints.

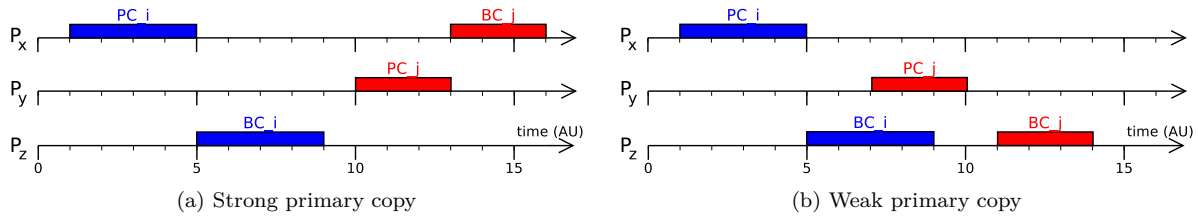
When carrying out simulations, dependent task were modelled by DAGs. It was found that the rejection ratio of dependent tasks is at least three times higher than that of independent jobs due to precedence constraint and that most tasks are rejected because backup copies cannot be scheduled before deadline. Moreover, the replication cost for DAGs is much higher than for independent tasks due to overloading restrictions. Last but not least, the more task dependencies, the worse the system performances in terms of rejection rate, replication cost and response time.

The authors also studied a scenario consisting of 40% of independent jobs and 60% of DAGs and, as expected, they show that its rejection rate is between that of independent and dependent tasks.

Another algorithm dynamically scheduling dependent tasks was presented in [160]. Since it is meant for virtualised clouds, which is one of the fields using the primary/backup approach, it is described in Section 2.7.3.

In [121], the authors defined the term "strong primary copy". In [160], the authors completed this definition with the term "weak primary copy". To explain these terms, we consider that a task t_j is dependent on task t_i. It means that the task t_i is a parent of the task t_j and the task t_j is a child of the task t_i.

The *strong primary copy* (PC) "is always executed if its processor is operational" [160], i.e. the finish time of backup copy of task $t_i(BC_i)$ is before the start time of PC_j. The *weak primary copy* "may not be executed even if its processor is operational" [160]. Its constraints are as follows: (i) the finish time of PC_i is before the start time of PC_j, (ii) the finish time of BC_i is before the start time of BC_j, and (iii) PC_i and BC_j cannot be mapped on the same processor. An example of strong and weak primary copies are depicted in Figures 2.10.



(a) Strong primary copy (b) Weak primary copy

Figure 2.10 – Difference between strong and weak primary copies

While there are no additional constraints when scheduling strong copies, there are some for weak copies. All aforementioned papers dealing with dependent tasks [121, 155, 160] study them. For example in [160], they introduced following notation:

— $\Delta_i\{.\}$: set of tasks causing a weak primary copy, i.e. the set of tasks that are parents of a task t_i and PC_i cannot receive messages from those task backup copies;
— $\Delta_i^P\{.\}$: set of primary copies of tasks causing a weak primary copy, i.e. those tasks that are in the set $\Delta_i\{.\}$;
— $PS\left(\Delta_i^P\{.\}\right)$: set of processors accommodating primary copies of tasks causing a weak primary copy, i.e. those tasks that are in the set $\Delta_i\{.\}$.

Based on these notations, we sum up the constraints in Table 2.1, where $\mathrm{P}(BC_j)$ denotes the processor on which a BC_j is mapped.

Table 2.1 – Constraints on mapping of primary copies of dependent tasks

PC_i	PC_j	**Constraints**
Strong	Strong	No
	Weak	$\mathrm{P}(BC_j) \neq \mathrm{P}(PC_i)$
Weak	Strong	No
	Weak	$\mathrm{P}(BC_j) \notin \mathrm{PS}(\Delta_i^P\{.\})$

As it can be seen, the management of precedence constraints is not straightforward in [121, 155, 160]. To avoid this complexity, R. Devaraj et al. suggested an offline method to assign tasks in DAG with individual deadlines [42]. Once each task has its own start time and deadline, it can be scheduled as an independent task. Their proposed method is presented in Algorithm 5. The authors do not mention a DAG arrival time in the algorithm. Therefore, to make the formulae more general, it would be necessary to add this time, for example $d_i = a_{DAG} +$ (computed deadline). The *source task* is the task without any predecessor and the *sink task* is a task without any successor.

Algorithm 5 Determination of start times and deadlines of tasks in DAG [42]

Input: Set of DAGs without assignation of start times and deadlines to tasks
Output: Set of DAGs with assignation of start times and deadlines to tasks

1: Sum execution times (E_{P_i}) of all tasks in each distinct path P_i from source task to sink task
2: Sort the paths in the non-increasing order of their E_{P_i}
3: **for** all paths **do**
4: **if** current path P_i contains a subset of tasks with already assigned deadlines **then**
5: Compute the sum of the deadlines (d_{path}) of all tasks with already assigned deadlines in P_i
6: Compute the sum of the execution times (E_{rem}) of tasks without already assigned deadlines
7: Assign deadlines to tasks without already assigned deadlines: $d_i = \lfloor \frac{E_i}{E_{rem}} \cdot (d_{DAG} - d_{path}) \rfloor$
8: **else**
9: Assign deadlines d_i to each task T_i with execution time E_i in P_i: $d_i = \lfloor \frac{E_i}{E_{P_i}} \cdot d_{DAG} \rfloor$

The authors state that their method is optimal with regard to the transformed tasks, i.e. deadlines are uniformly distributed weighted by computation times. Nevertheless, since every task has its own individual deadline, the schedule of such a DAG may be suboptimal when compared to the DAG containing tasks without individual deadlines.

2.6.1 Experimental Framework

In Table 2.3, we compare the experimental frameworks of several papers. In general, they consist of the directed acyclic graphs (DAGs) characterised by the arrival time and deadline, and containing several

tasks. These DAGs are then scheduled on heterogeneous processors. The parameters to generate DAGs, such as the number of tasks and their characteristics, are then summed up in Table 2.3.

Table 2.2 – Simulation parameters for dependent tasks modelled by DAGs

Parameter	[121]	[155]	[160]	[99]
# simulations	?	25	?	50
# DAGs	100 000	100 000/simulation	$(50; 300)$	50/simulation
DAG arrival time t_a	?	Poisson process with rate λ	Poisson distribution with the average interval time $1/\lambda$ being uniformly distributed in $\left(\frac{1}{\lambda}; \frac{1}{\lambda+2}\right)$; $\frac{1}{\lambda} \in [0; 10]$	?
DAG deadline t_d	?	?	$d_i = a_i + \alpha \cdot e_i^{min}$ where e_i^{min}: minimum possible DAG execution time and $\alpha \in [1.5; 2.5]$	?
DAG execution time t_{exe}	?	Exponential with a mean of $\frac{1}{\mu}$	-	-
# processors	9; 10; 12; 16	16; 80; 400; 2000	-	8; 16; 32; 64; 128
Processor heterogeneity	Yes	Uniform $(1.0; 10.0)$	-	Yes
Computational time	Defined by execution time for each processor	$\frac{\text{Execution time}}{\text{Processor heterogeneity}}$	-	Defined by computation cost matrix
Host processing capacity	-	-	1000; 1500; 2000; 3000 MIPS	-
VM processing power	-	-	250; 500; 700; 1000 MIPS	-

2.6.2 Generation of DAGs

To carry out the simulations and evaluate the algorithm performances, there are two possibilities of obtaining the input data: either tasks dependencies are already available since they stem from real applications or they need to be synthetically created. The most commonly used tools to generate a synthetic directed acyclic graph (DAG) are as reads:

— **DAGGEN**
This tool [1] generates random synthetic task graphs and was designed to evaluate scheduling algorithms. The last version dates back to 2013 and it was used for example in [97]. The merits are related to its easy utilisation and possibility to set different graph parameters, which are presented in more detail in Appendix B.
An example of generated code to model a DAG is given in Figure 2.11.

— **Task Graph For Free (TGFF)**
This graph generator [2] was developed by K. Vallerio, D. Rhodes and R. P. Dick. The last version dates back to 2008. The TGFF generates pseudo-random graphs for use in scheduling and allocation research.
Each graph consists of nodes and edges and it is assigned a period and a deadline based on the length of the maximum path in the graph and the *task_trans_time*, which is the average time per

1. `https://github.com/frs69wq/daggen`
2. `https://robertdick.org/projects/tgff/index.html` and `http://ziyang.eecs.umich.edu/~dickrp/`

Table 2.3 – DAG parameters

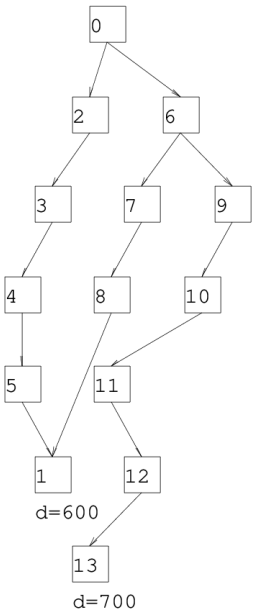
Parameter	[121]	[155]	[160]	[99]
# tasks N	100; 200	20, 40, 60, 80, 100	200	500; 1000; 1500; 2000; 2500
# messages U	$4N$	-	$\theta \cdot N$; $\theta \in \{2; 7\}$	-
Connectivity	-	Randomly chosen; uniform $(1\%; 100\%)$ (fully connected)	-	$\lambda \in \{0.2; 0.5; 1; 2; 5\}$
# levels	-	-	-	$\lceil \frac{\sqrt{N}}{\lambda} \rceil$
Width	-	-	-	$\lceil \lambda \cdot \sqrt{N} \rceil$
Communication to computation cost ratio (CCR)	-	-	-	0.2; 0.5; 1; 2; 5
Communication time for each message	Randomly selecting a sender and a receiver for each edge having cost [1; 10]	-	Randomly selecting a sender and a receiver for each message having size $[10; 100]$ MB	CCR ·(average task execution time)
Task execution time	Random $(5; 50)$	Uniformly distributed with a mean of $\frac{t_{exe}}{N}$	-	Uniform $(10; 50)$
Task size	-	-	Uniform $\left(1 \cdot 10^5 \text{ to } 2 \cdot 10^5\right)$ MI (Millions of Instructions)	-
Fault detection time δ	Randomly chosen; uniform $(1; 10)$	-	-	-
Relative deadline t	Depends on the task constraints	Uniformly distributed with a mean of $t_a + \eta \cdot \frac{2 \cdot t_{exe}}{\text{mean processing speed}}$; $\eta \in \{0.2; 0.3\}$	According to the DAG deadline	-

```
// DAG automatically generated by daggen at Fri Jun  8 14:35:27 2018
// ./daggen -n 10 --maxdata 10000 --dot -o DAG_name.dot
digraph G {
  1 [size="437468061946", alpha="0.04"]
  1 -> 6 [size ="679477248"]
  2 [size="13268109502", alpha="0.04"]
  2 -> 4 [size ="536870912"]
  2 -> 5 [size ="536870912"]
  2 -> 6 [size ="536870912"]
  3 [size="11659573117", alpha="0.18"]
  3 -> 4 [size ="411041792"]
  4 [size="549755813888", alpha="0.19"]
  4 -> 7 [size ="536870912"]
...
```

Figure 2.11 – Example of DAG generation using DAGGEN

node and edge traversal. A user can set several parameters, such as general ones (e.g. the number of graphs to generate or the minimum number of tasks per task graph), serial/parallel ones (e.g. the length and the width of series chains, the Boolean to generate a graph with a series-parallel

structure or the Boolean to force all paths to rejoin the last node) and other parameters (e.g. the probability that a deadline is hard or the laxity of periods relative to deadlines, i.e. how deadlines are respected).
An example of DAG generation is illustrated in Figures 2.12. A generated DAG is depicted in Figure 2.12a. The task dependencies and task and edge types are summed up in Figure 2.12b. These types are then characterised in Figure 2.12c.



(a) DAG

```
@TASK_GRAPH 0 {
        APERIODIC

        TASK t0_0       TYPE 2
        TASK t0_1       TYPE 3
        TASK t0_2       TYPE 3
        TASK t0_3       TYPE 3
        TASK t0_4       TYPE 2
        TASK t0_5       TYPE 1
        TASK t0_6       TYPE 0
        TASK t0_7       TYPE 4
        TASK t0_8       TYPE 1
        TASK t0_9       TYPE 1
        TASK t0_10      TYPE 0
        TASK t0_11      TYPE 1
        TASK t0_12      TYPE 2
        TASK t0_13      TYPE 1

        ARC a0_0        FROM t0_0  TO  t0_2 TYPE 0
        ARC a0_1        FROM t0_2  TO  t0_3 TYPE 0
        ARC a0_2        FROM t0_3  TO  t0_4 TYPE 0
        ARC a0_3        FROM t0_4  TO  t0_5 TYPE 0
        ARC a0_4        FROM t0_5  TO  t0_1 TYPE 0
        ARC a0_5        FROM t0_0  TO  t0_6 TYPE 0
        ARC a0_6        FROM t0_6  TO  t0_7 TYPE 0
        ARC a0_7        FROM t0_7  TO  t0_8 TYPE 0
        ARC a0_8        FROM t0_8  TO  t0_1 TYPE 0
        ARC a0_9        FROM t0_6  TO  t0_9 TYPE 0
        ARC a0_10       FROM t0_9  TO  t0_10 TYPE 0
        ARC a0_11       FROM t0_10  TO  t0_11 TYPE 0
        ARC a0_12       FROM t0_11  TO  t0_12 TYPE 0
        ARC a0_13       FROM t0_12  TO  t0_13 TYPE 0

        HARD_DEADLINE d0_0 ON t0_1 AT 600
        HARD_DEADLINE d0_1 ON t0_13 AT 700
}
```

(b) Task dependencies

```
@TASK_COMPUTATION_TIMES 0 {
# price
  6.50996

#-------------------------------
# type version computation_time
  0    0       5.43047
  1    0       9.1229
  2    0       11.3788
  3    0       7.48397
  4    0       14.2219
}
```

(c) Types characteristics

Figure 2.12 – Example of DAG generation using the TGFF

The TGFF is suitable for applications requiring generation of the pseudo-random graphs. Its merits are mainly the possibility to create task dependencies and manage different task and edge types. The drawbacks are related to the parameter *task_trans_time*. This parameter can be set at an average value only, which means that all tasks have the same value. Moreover, the deadlines depend on this parameter. In addition, according to [39], there is no way to control the random distribution of the attributes generated by TGFF.

— **Synchronous Dataflow (SDF)**
The toolkit [3] SDF3 was developed by S. Stuijk, M. Geilen and T. Basten from the University of Eindhoven [140]. The SDF3 is not only a random generator of synchronous dataflow graphs but it is also capable to carry out transformations and analysis of synchronous dataflow graphs. The current version dates back to 2014.

— **GGen**
The GGen [4] is another tool to generate and analyse DAGs. The user chooses a method how a new DAG is created. He or she can select, for example the Erdős-Rényi methods ($G(n,p)$ or $G(n,M)$), layer-by-layer, fan-in/fan-out, or random orders, which are described in [39].

— **P-method**
The P-method to generate random task graph was described in [7]. It is based on the probabilistic construction of a Boolean adjacency matrix ($m \times m$) using a Bernoulli process. If a matrix element

3. http://www.es.ele.tue.nl/sdf3/
4. https://github.com/cordeiro/ggen

a_{ij} (where $0 \leqslant i \leqslant m$ and $0 \leqslant j \leqslant m$) equals 1, there is a dependency from task t_i to task t_j. Otherwise, there is no dependency.

If a tool does not provide a graphical visualisation, once a DAG is generated, it can be treated by Graphviz [5]. This tool allows one to obtain a graphical representation in different formats, such as image or PDF files. Graphviz makes use of the DOT language.

2.7 Application of Primary/Backup Approach

The primary/backup approach is a simple method to make system fault tolerant. This section presents several examples, where this approach is successfully used. The use in a system based on dynamic voltage and frequency scaling is covered in Section 2.7.1. Section 2.7.2 then describes how this approach is put into practice in evolutionary algorithms. The application in virtualised clouds and in satellites are respectively treated in Sections 2.7.3 and 2.7.4.

2.7.1 Dynamic Voltage and Frequency Scaling

The paper [67] introduces several algorithms based on the primary/backup approach to schedule independent periodic real-time tasks on a multiprocessor system using both the *dynamic voltage and frequency scaling* (DVFS) and the *dynamic power management* (DPM). The aim is to maximize the energy savings subject to the constraints of (i) tolerating a single permanent fault and (ii) preserving system reliability with respect to transient faults (in the absence of permanent faults).

Before describing the algorithms, we briefly summarise the system and task models. The authors consider m homogeneous processors with shared memory and they assume that each processor has dynamic voltage and frequency scaling (DVFS) capability, i.e. it can operate at one of several discrete frequency and voltage levels.

The system has a set of n independent periodic real-time tasks $\Gamma = \{T_1, ..., T_n\}$. Each task T_i is characterised by its worst-case execution time c_i and its period p_i. The authors consider that the worst-case execution time c_i corresponds to the execution time at the maximum available processor frequency. The tasks are assumed to have implicit deadlines: a j^{th} task instance of T_i denoted as $T_{i,j}$ arrives at time $(j-1) \cdot p_i$ and needs to complete its execution by its deadline at $j \cdot p_i$.

The power consumption of a system with m processors operating respectively at frequencies $f_1, ..., f_m$ is expressed as follows:

$$P(f_1, \ldots, f_m) = P_s + \sum_{i=1}^{m} \hbar_i \left(P_{ind} + C_{ef} \cdot f_i^k\right) \tag{2.2}$$

where

- P_s denotes the system static power,
- P_{ind} stands for the frequency-independent active power (assumed to be the same for all processors),
- the product $C_{ef} \cdot f_i^k$ stands for the frequency-dependent active power depending on the system-dependent constants C_{ef} and k and the frequency f_i,
- $\hbar_i$ is Boolean: if a i^{th} processor is active, $\hbar_i = 1$, otherwise $\hbar_i = 0$. It means that the processor is switched to the sleep state through the dynamic power management (DPM) and does not consume any active power.

Following the model definitions, we describe the algorithms.

First, two algorithms based on *Standby-Sparing* (SS) scheme are introduced: *Paired-SS* and *Generalized-SS* algorithms. In general, the SS scheme schedules offline primary and backup copies separately on the primary and backup processors. Since the backup copies are normally deallocated once the corresponding

5. `https://www.graphviz.org`

primary copies are correctly executed, the backup copies are scheduled as late as possible. Consequently, the algorithms based on the SS scheme execute primary copies early at scaled frequency and backup copies as late as possible at the maximum frequency.

The Paired-SS algorithm couples processors to pairs and tasks are separately scheduled for each processor pair. To generalise this algorithm, the authors consider the Generalized-SS algorithm, which divides processors into primary and secondary processor groups (of potentially different sizes) and then schedules primary (backup) tasks on the primary (secondary) processors, respectively. In order to save energy, primary copies are scheduled using the *earliest deadline first* (EDF) algorithm and the DVFS technique, while backup copies are scheduled making use of the *earliest deadline latest* (EDL) algorithm and the DPM technique.

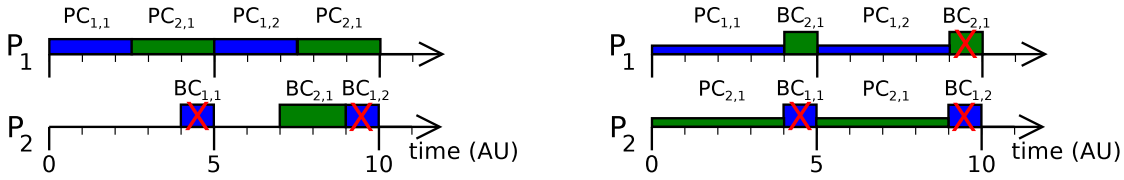
Second, the authors studied the scheduling of the primary and backup copies in a mixed manner on all processors. Although this choice increases the scheduling complexity because the copies are not allocated to their dedicated processors, it makes the use of the slack easier for more energy savings.

The primary copies are scheduled using the *preference-oriented earliest deadline* (POED) algorithm, which chooses whether a copy is scheduled as soon as possible (ASAP) or as late as possible (ALAP) depending on the task priority. The earlier the deadline, the higher the priority.

Regarding the backup copies, there are two processor allocation policies. If the workload after placing primary copies is balanced, the *cyclic backup allocation* is put into practice. The backup copies are allocated only to neighbour processors where their primary copies are scheduled. If a primary copy is scheduled on processor P_i, the corresponding backup copy is placed on processor P_{i+1}. Otherwise, the *mixed backup allocation* is employed, which means that a backup copy can be scheduled on any processor in order to balance the workload, except the processor where the corresponding primary copy is scheduled.

Once the primary and backup copies are scheduled (considering that their execution is carried out on the maximum frequency), the algorithm checks all slacks in order to scale processor frequencies when executing primary copies. All backup copies are planned to be executed at the maximum processor frequency because they will be deallocated if no fault occurs.

To illustrate the difference between a Standby-Sparing and Preference-Oriented Earliest Deadline algorithms, we plot Figures 2.13 illustrating an example of schedules for a 2-processor system with two periodic tasks T_1 and T_2 characterised respectively by $c_1 = 1$, $p_1 = 5$, $c_2 = 2$ and $p_2 = 10$. According to the Standby-Sparing algorithm (Figure 2.13a), the primary copies are scheduled under the earliest deadline first policy and executed at the scaled frequency of 0.4. The backup copies are placed under the earliest deadline latest policy and executed at the maximum frequency. As for the Preference-Oriented Earliest Deadline algorithm (Figure 2.13b), the primary copies are executed at the scaled frequency of 0.25 and the backup copies are run at the maximum frequency. Once results of the primary copies are available and error-free, the corresponding backup copies can be deallocated (as illustrated by red crosses). A similar idea was presented in [70].



(a) Standby-Sparing algorithm

(b) Preference-Oriented Earliest Deadline algorithm

Figure 2.13 – Schedules generated by two algorithms using different allocation policies[6](Adapted from [67, Figure 2])

6. The height of blocs representing the task copies is proportional to the frequency at which they are executed.

To sum up the Preference-Oriented Earliest Deadline (POED) algorithm schedules the primary copies executed at scaled frequencies as soon as possible (ASAP) and the backup copies executed at the maximum frequency as late as possible (ALAP) to save energy. The four main steps are as follows:

1. Allocate primary copies (offline)
2. Allocate backup copies (offline)
3. Calculate scaled frequencies for primary copies (offline)
4. Execute tasks considering ASAP preference for primary copies and ALAP preference for backup copies (can be adapted online)

The authors presented algorithms using the DVFS technique even if they mention in their paper that, despite the effectiveness of DVFS on reducing energy consumption, the DVFS has a negative effect on the system reliability due to the significantly increased transient fault rates at low supply voltages. This confirms our conclusion made in Section 1.5 and based on the previous work related to DVFS.

They carried out experiments with a 16-processor system and they found out that the Generalized-SS algorithm with different number of processors in primary and backup groups have better energy savings than the Paired-SS algorithm. Then, they showed that the POED-based algorithm generally performs better than SS-based algorithms and achieves better energy savings. In particular, the online technique takes advantage of all available slack (due to the backup deallocation or quicker task execution) and the algorithm can slow down the execution of primary copies and/or delay the execution of backup copies, which results in much reduced overlapped executions and therefore less energy consumption.

2.7.2 Evolutionary Algorithms

Kumar et al. [87] study the fault tolerant scheduling making use of the primary/backup approach. Their proposed algorithms employ the genetic algorithm (GA) based on evolutionary computing technique and/or the ant colony optimisation algorithm (ACOA) using computational models inspired by the collective foraging behaviour of ants. The paper [126] from the same lead author presents the algorithm based on the particle swarm optimisation (PSO), which is inspired by the natural phenomenon of social interaction and communication, such as bird flocking and fish schooling. The main aim of all mentioned algorithm is to minimise the makespan, i.e. the completion time of the last task.

The system consists of m identical processors connected through the shared memory and it deals with n independent aperiodic hard real-time tasks, which are assumed to be non-preemptible and non-parallelizable. Every task is modelled by four parameters: arrival time, ready time, worst-case execution time and relative deadline.

Regarding the fault model, independent faults are permanent or transient and only one fault may occur in the system at the same time. Moreover, only hardware faults are considered and all processors have equal chance of fault occurrence.

The traditional fault tolerant scheduling (TFTS), as well as their proposed algorithms, schedule primary copies as soon as possible and backup ones as late as possible taking into account the mutual exclusion of copies in space and time. They explicitly make use of the backup-backup overlapping and implicitly of the backup deallocation.

In general, evolutionary algorithms, to which the GA, the ACOA and the PSO belong, represent a group of population-based black-box metaheuristic optimisation techniques that provide quasi optimal solution to complex NP-hard problems without any domain specific knowledge. Although any prior knowledge of the problem characteristics is not required, the algorithms need to hybridise with other techniques or knowledge to enhance its performance and they are usually implemented based on five key elements: solution representation, initial population, fitness function, algorithm specific manipulative operations and hybridisation with domain specific knowledge.

The evolutionary algorithms consist of two main phases: initialisation and iteration. During the initialisation, the algorithm randomly generates an initial population of fixed size Np. Next, each iteration

creates a new population from the best solution of the old population by appropriate manipulative operations. The second phase, i.e. the iterations, continues until the termination criterion is met, which corresponds to the global optimum solution.

In the case of the primary/backup approach scheduling, five key elements are as follows:

— **Solution representation**
The authors propose to represent a solution as a schedule of length n, where n is the number of tasks in the task set. Every solution, also called the *solution vector*, S_i is modelled by a sequence of n task tuples of the form $\{(T_i, P_i, B_i)\}$, as depicted in Figure 2.14. For instance, a tuple $\{(2, 4, 1)\}$ represents the task number 2 having its primary copy mapped on processor 4 and the backup one on processor 1. These tuples are arranged according to the scheduling order, which means that the earlier the tuple is situated, the sooner the corresponding task is treated.

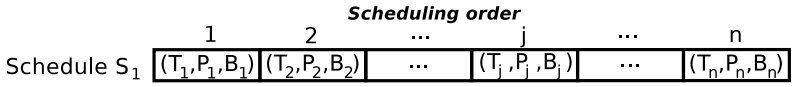


Figure 2.14 – Structure of the solution vector (Adapted from [126, Figure 1])

— **Initial population**
The initial population of individuals is generated as a matrix $(Np \times n)$ composed of Np solution vectors. The value of each component of the task tuple is assigned a random number within their respective permissible range satisfying restrictions.
At every iteration, a new population of Np solutions is created, the *fitness function* (for GA and PSO) and the *processor and task status* (PATS) record (for ACOA) for each schedule S_i are evaluated. Figure 2.15 illustrates a structure of a population of Np individuals.

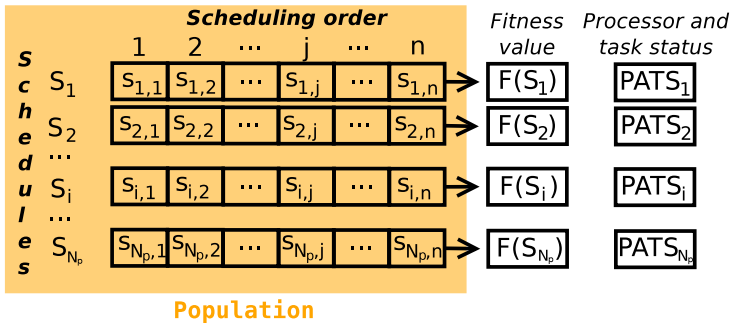


Figure 2.15 – Structure of the population (Adapted from [87, Figure 3])

— **Hybridisation with domain specific knowledge**
From the viewpoint of the ACOA and GA, a schedule S_i is assimilated to an ant and characterised by two factors: the *fitness value* $F(S_i)$ and the *Total Pheromone Intensity*, which is the sum of pheromone intensities at each task tuple in S_i and which is involved in the PATS record. The pheromone intensity is related to the ratio of the number of successfully scheduled tasks to the number of rejected tasks and the evaporation and deposition in the course of time.
When considering the PSO, a solution space is represented by a swarm of birds called the *particles*. Each particle accounts for a candidate solution to the problem and is characterised by its position vector $\mathbf{x}$ and velocity vector $\mathbf{v}$.

— **Algorithm specific manipulative operations**
The proposed algorithms employ basic genetic operations, such as selection, crossover and mutation, to evolve existing population to a new generation without violating aforementioned assumptions of the primary/backup approach. The *selection* provides stable and fast convergence, the *crossover* advances exploration capability and the *mutation* brings diversity in the population.

— **Fitness function**
The fitness function of solution S_i is in general defined as follows:

$$F(S_i) = \lambda_1 \cdot f_1 + \lambda_2 \cdot f_2 + \lambda_3 \cdot f_3$$

where
— λ_j is a relative weight factor, such as

$$\lambda_1 + \lambda_2 + \lambda_3 = 1$$

— $f_j \in [0; 1]$ is a cost component.
In the studied case, the values of λ_j are experimentally set at 0.7, 0.2 and 0.1 and the cost components are associated with the following metrics:
— f_1= rejection ratio $= \frac{\text{number of rejected tasks}}{\text{number of submitted tasks}}$
— f_2= processor utilisation deviation ratio $= \frac{\text{standard deviation in processor utilisation}}{\text{standard deviation in processor load}}$
— f_3= earliest finishing time ratio $= \frac{\text{earliest finishing time of the last task in } S_i}{\text{maximum absolute deadline among all the tasks in } S_i}$
In order to ensure that the cost components data are measured on a neutral scale, all values of fitness functions are normalised to [0,1].

The main scheduling steps of the traditional fault tolerant scheduling are as follows:

1. Search for a primary copy slot as soon as possible,
2. Search for a backup copy slot as late as possible,
3. If PC and BC slots exist, commit the task.

Regarding the main scheduling stages for algorithms based on the GA, ACOA or PSO, they are as reads:

1. *Setup*: if a slot is not empty, remove the first tuple from it,
2. *Discard and repeat*: check if time constraints are satisfied,
3. Check primary copy slot from the current tuple,
4. Check backup copy slot from the current tuple,
5. If PC and BC slots exist, commit the task.

The simulations being carried out in Matlab, the authors consider 4, 8, 12 and 16 processors. The algorithm parameter Np equals 100 all the time and there are 200, 300 or 500 iterations depending on the size of task set varying 10 to 100. In fact, the population size should be fixed at a reasonably high value ($Np = 100$) in order to ensure the output stability and the convergence. The TFTS is run without faults first and then with fault injection. The algorithms making use of the GA, ACOA or PSO are simulated with fault injection only. One simulation scenario is run 40 times and obtained values are then averaged.

The results are evaluated by means of both the rejection rate and fitness function represented as a function of the number of executed iterations. The simulations show that the scheduling based on the GA, ACOA or PSO outperform the TFTS. The scheduling based on the GA has faster convergence but is slower when compared to the scheduling with the ACOA. The algorithm using the PSO shows uniformity in processor utilisation in comparison to the TFTS. Moreover, it can be seen that at least 8 iterations are required to schedule 10 tasks on 4 processors and at least 50 iterations (for GA) or 200 iterations (for ACOA) or 250 iterations (for PSO) are necessary to place 100 tasks on 16 processors.

To sum up, the GA, ACOA and PSO present an interesting implementation for the conventional primary/backup approach. On the one hand, as the GA brings the genetic operations, such as the selection, crossover and mutation, and the ACOA fetches the social behaviour, they can avoid that the scheduling algorithm gets stuck in a local optimum and does not converge. On the other hand, it seems that the use of such techniques is not suitable for systems dealing with the hard real-time tasks because the presented algorithm requires many computations even for a small amount of tasks. Actually, the higher the number of tasks and the lower the number of processors, the slower the convergence.

2.7.3 Virtualised Clouds

Zhu et al. [160] make use of the primary-backup approach for the virtualised cloud by taking into account cloud characteristics. First, the cloud uses virtual machines as basic computational instances and allows them to migrate among multiple hosts. Second, the cloud can be scaled up and down depending on the demand. The authors propose the fault tolerant algorithm, which schedules dependent tasks on the cloud and which can add or remove resources according to the workload.

Jobs consisting of dependent tasks are modelled by the Directed Acyclic Graph (DAG) denoted by $G = \{T, E\}$, where $T = \{t_1, t_2, ..., t_n\}$ is a set of the real-time non-preemptive tasks and E is a set of the directed edges that represents dependencies among tasks. Every DAG is defined by arrival time and deadline. Each task in the DAG is characterised by arrival time, deadline and task size.

The authors consider a virtualised cloud containing a set $H = \{h_1, h_2, ...\}$ of unlimited number of physical computing hosts. A host $h_k \in H$ has its processing capacity p_k, which characterises its CPU performance in Million Instructions Per Second (MIPS), and it can have several virtual machines. Its virtual machines represent a set $V_k = \{v_{1k}, v_{2k}, ...\}$ and they can have different processing abilities whose sum is at most equal to p_k.

Regarding the fault model, the system deals with independent faults, which can be transient or permanent, and it makes use of a fault-detection mechanism to detect faults. There is at most one host failure at the same time.

The authors propose the dynamic **f**ault toler**a**nt **s**cheduling algorithm for real-**t**ime sci**e**ntific wo**r**kflows, called FASTER that is responsible not only for the scheduling of DAGs but also for the elastic resource provisioning. The algorithm processes DAGs, also called workflows, in order of their arrival and it searches for a primary copy schedule first and then for a backup copy schedule. The primary and backup copies of the same task can be executed in parallel. Furthermore, if one task in the DAG misses its deadline, the DAG is not rejected because its deadline may still be met. If two or more tasks miss their deadlines, the DAG is rejected and all its reserved resources are reclaimed.

The first part of the FASTER schedules the task copies. The primary copies are placed as soon as possible and the search for free slots starts on hosts having only a few primary copies. The idea is to have an even distribution of primary copies over all the active hosts in order to increase the possibility of primary-backup overlapping, which is the same as the primary-backup overloading.

This paper shows that the weak primary copy [7] has more scheduling constraints than the strong primary copy and the main reason for a primary copy to become weak is that it cannot receive results from its predecessor before its start time if a fault occurs. To reduce this phenomenon, the algorithm thus schedules the backup copies as soon as possible and starts the search on host already accommodating a lot of backup copies. In fact, if the fault occurrence is rather rare, the primary copies are correctly executed and groups of backup copies are progressively deallocated, which can completely free a host that can be then switched off. The proposed algorithm does not make use of the backup-backup overlapping.

In order to increase the system schedulability, the authors put into practice the *backward time slack*, which indicates how long the start time of a task copy can be shifted backward without any impact on the start time and the status (i.e., strong or weak primary copies) of the subsequent tasks. Besides, they employ a reclamation mechanism, which finishes the execution of the backup copy and frees its reserved slot, if the corresponding primary copy is correctly executed.

The second part of the proposed algorithm deals with the cloud elasticity. It means that, when the system is charged, the algorithm adds resources, i.e. scales up, to avoid the task rejection and, when the workload is lower, it turns off resources, i.e. scales down, if they have not been used for a certain amount of time.

There are two possibilities of scaling: the vertical and the horizontal ones. The former one creates or removes a new virtual machine with the required processing capability and the latter one increases or shrinks the processing capacity of an existing virtual machine.

7. The definitions of "weak" and "strong" primary copies are given in Section 2.6.

To measure the algorithm performances, three following metrics are put into practice:
— the *guarantee ratio* (GR) accounting for the percentage of DAGs that are guaranteed to finish successfully among all submitted DAGs,
— the *host active time* (HAT) standing for the total active time of all hosts in cloud and thus informing about the system resource consumption,
— the *ratio of task time over hosts time* (RTH), which is the ratio of the sum of the task execution times to the sum of the host active times and which reflects the system resource utilisation of the system.

The results show that when the number of DAGs increases, the guarantee ratio remains the same because the system can dynamically launches new resources. Consequently, the system resource consumption and utilisation grow. When tasks in the DAG become more dependent, all three studied metrics slightly decrease because the possibility of executing tasks in parallel declines.

Moreover, when the interarrival time increases, the system becomes less charged, the system resource consumption and utilisation remain almost constant because the system can adjust the number of resources. Nevertheless, the guarantee ratio increases for the creation of new resources introduces a delay which may cause deadline misses of several tasks.

In addition, when the deadline becomes more tight, the guarantee ratio rapidly decreases because the task deadlines can be missed due to the time required to launch new resources. For that reason, the system resource consumption and utilisation drop as well.

To summarise, the authors present the algorithm that can efficiently schedule dependent tasks at run-time and dynamically add or remove resources depending on the demand. The algorithm improves the system schedulability and resource utilisation. Nonetheless, these merits are at the expense of the algorithm complexity, which is not studied in this paper. Furthermore, when Naedele [104] carried out the experiments with independent tasks only, it found out that the results using the slack, which is equivalent to the task backward shifting, may be sometimes worse than the ones without this technique.

2.7.4 Satellites

Zhu et al. [161] present an enhancement based on the primary/backup approach to provide satellites with fault tolerance. They propose the *Fault-Tolerant Satellite Scheduling* (FTSS) algorithm to dynamically schedule aperiodic, real-time, independent and non-preemptive tasks. In order to improve the resource utilisation, the algorithm makes use of the overlapping, which is the same as the overloading described in [61]. Nevertheless, the technique of backup deallocation is not mentioned in this paper and thus not employed.

The task is modelled by the arrival time, deadline and resolution requirement, which corresponds to the worst acceptable resolution. Every task has two identical copies: the primary and backup ones. Each copy executes on different satellites. Actually, every satellite has one processor and the fault tolerance is therefore ensured by a set of satellites. The satellite is characterised by the duration of task execution, field of view angle, slewing pace, start-up time, retention time of shutdown, attitude stability time, maximum slewing angle and the best ground observation resolution. The communication and task dispatching times are not considered.

The authors then define the *available opportunity* as a possible slot for a task copy if all time and resolution constraints are fulfilled. An example of the k^{th} available opportunity ao^P_{ijk} for the primary copy of task T_i on satellite S_j, delimited by the difference between the end PC window we^P_{ijk} and the start PC window ws^P_{ijk}, is depicted in Figure 2.16. The area captured by satellite S_i at the beginning of the available opportunity is coloured in red and purple, whereas the one captured at the end of the available opportunity is highlighted in purple and blue. The purple zone illustrates the area where the task T_i should be in order to be visible during the whole available opportunity.

The fault model considers transient or permanent faults assumed independent. Furthermore, only one fault can occur at the same time and a fault detection mechanism is available to detect it.

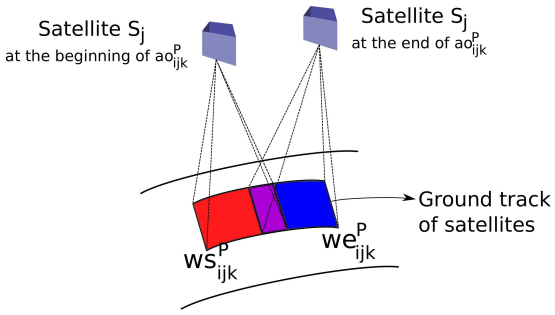


Figure 2.16 – Example of the k^{th} available opportunity ao_{ijk}^{P} for the primary copy of task T_i on satellite S_j (Adapted from [161, Figure 1])

The main objective of the presented algorithm is to maximise the guarantee ratio and then to minimize the observation resolutions of all accepted tasks under time constraints.

The algorithm schedules the primary copies as soon as possible and the backup ones as late as possible after scouring for available opportunities on all processors to find the best solution in a sense of the main objectives. To enhance the algorithm performances, the backup-backup overlapping, described in [61], and the primary-backup overlapping, presented in [6], are implemented. Considering that a task T_i arrives earlier than a task T_j, the authors point out that the latter technique cannot be used for PC_j and BC_i if the start time of PB_j is earlier than the one of BC_i because PC_j cannot be interrupted during its execution.

In addition, taking into account that a scene can be seen by several satellites at the same time (but with different properties, such as the resolution or angle), it is possible to merge some tasks together in order to improve the schedulability. Consequently, when an image is the same for two different primary copies, these copies can jointly merge if several merging constraints on the existence and size of overlapped time window, the observation angle and the resolution are met.

The simulation parameters are set at the following values: the latitude between -30°and 60°, the longitude between 0°and 150°, the number of tasks in range from 200 to 1 200 and the number of satellites at 10. The interarrival time is uniformly distributed. To assess the algorithm performances, the authors evaluate the *guarantee ratio* and the *observation resolution* accounting for the average observation resolution of accepted tasks.

Their experiments compare the proposed algorithm with the basic one (no merging and no overlapping), the one with merging only and the one with overlapping only. It can be observed that the use of the task merging and overlapping together generally achieves better results, i.e. higher guarantee ratio and lower observation resolution. Besides, the more tasks, the higher observation resolution and the worse the guarantee ratio due to higher system workload. In general, the longer the interarrival time, the higher the guarantee ratio because they are less arriving tasks. Nevertheless, when the interarrival time is too short, the values of the guarantee ratio are comparable with the ones when the interarrival time is high for many tasks can be merged. Moreover, it is shown that the tighter the task deadline, the lower the guarantee ratio.

2.8 Summary

This chapter summed up the work related to the primary/backup approach. It covers the advent of this approach and already proposed enhancing techniques: the primary slack, decision deadline, active approach, replication cost with boundary schedules and primary-backup overloading. It also showed several applications of this approach, such as its use in the dynamic voltage and frequency scaling, evolutionary algorithms, virtualised clouds, or satellites.

While this chapter dealt with already publish work, the next chapter presents our research on the primary/backup approach.

Chapter 3

Primary/Backup Approach: Our Analysis

The preceding chapter is a compilation based on already published sources related to the primary/backup approach. The first part of this chapter is devoted to independent tasks, whereas the second one treats dependent tasks.

This chapter presents our task, system and fault models. Following the mathematical problem formulation, different processor allocation policies and scheduling search techniques are compared. Next, three proposed enhancing techniques are introduced: (i) the method of *restricted scheduling windows* within which the primary and backup copies can be scheduled, (ii) the method of *limitation on the number of comparisons*, accounting for the algorithm run-time, when scheduling a task on a system, and (iii) the method of *several scheduling attempts*. Finally, the experimental framework is described and results are analysed in fault-free and harsh environments.

Regarding the dependent tasks, this chapter presents how we deal with directed acyclic graphics and which aforementioned techniques are put into practice. The results are then described and discussed.

3.1 Independent Tasks

This section covers independent tasks, i.e. there exist no task dependencies.

3.1.1 Assumptions and Scheduling Model

A hard real-time system is composed of P interconnected identical processors sharing the same memory. Although the system with only homogeneous processors is considered, it would be possible to extend this model to a system with heterogeneous processors, such as in [155] by defining different processor speeds or different computation times. While a centralised memory is put into practice, a distributed memory could be used as well. The principle of the studied method would remain the same but it would necessitate to take delays of data transfers into account.

The aperiodic tasks are online scheduled on such a system without preemption. We assume the existence of fault detection mechanism and that it can promptly inform if a permanent and/or transient fault occurs. A fault can be detected for example by acceptance tests, such as timing, coding, reasonableness or structural checks [49].

We consider that only one processor failure can occur at any instant of time and that the scheduler is enough robust, e.g. using a spare scheduler if necessary. Current processors have a failure rate of $1/120\,h^{-1}$ [47] and, although the reliability of P-processor system is lower, our assumption holds. In fact, the authors in [72] proved that a system consisting of identical processors has its reliability (measured by means of the mean time between faults (MTBF)) equal to the processor MTBF divided by the number of processors. (For more details see Section 1.3.4.)

Using Graham's classification [66] described in Section 1.1, the analysed problem is defined as

$$P; m \mid n = k;\ online\ r_j;\ d_j = d;\ p_j = p \mid \text{(check the feasibility of schedule)}$$

Table 3.1 – Notations and definitions

Notation	Definition
a_i	Arrival time of task t_i
c_i	Computation time of task t_i
d_i	Deadline of task t_i
tw_i	Task window of task t_i
α	Multiple of c_i to define the size of task window
f	Fraction of task window tw_i
s_i	Slack of task t_i
ps_i	Percentage of s_i within the tw_i
PC_i	Primary copy of task t_i
BC_i	Backup copy of task t_i
xC_i	PC or BC of task t_i
$start(xC_i)$	Start of the execution of PC_i or BC_i
$end(xC_i)$	End of the execution of PC_i or BC_i

which means that k independent jobs/tasks (characterised by release time r_j, processing time p_j and deadline d_j) arrive online on a system consisting of m parallel identical machines and are scheduled to verify the feasibility of a schedule.

Regarding our task model, we assume that each task has three attributes: arrival time a_i, computation time c_i and deadline d_i. The task window tw_i is thus defined as $d_i - a_i$ and it can be also expressed as a multiple α of the computation time c_i. Since all task characteristics are known at the task arrival, our scheduling algorithm is online clairvoyant.

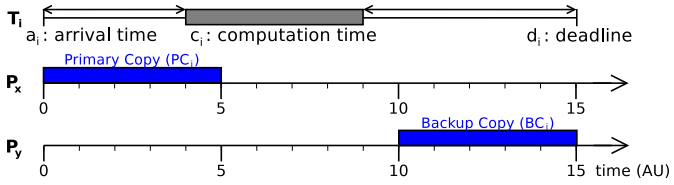


Figure 3.1 – Principle of the primary/backup approach

The studied algorithm is based on the *primary/backup (PB) approach* [61], which is commonly used for its minimal resource utilisation and high reliability and which was presented in Section 2.2. Its principal rule is that, when a task arrives, two identical copies, the *primary copy* (PC) and the *backup copy* (BC), are created. An example is illustrated in Figure 3.1. The primary copy is scheduled as soon as possible (ASAP) and the backup one as late as possible (ALAP) in order to avoid idle processors just after the task arrival time and possible high processor load later. A *slot* is a time interval on a processor schedule.

In order to improve the schedulability and minimise the resource utilisation, we consider the backup copy deallocation and the backup copy overloading, as introduced in Section 2.2.

Definition 1 (Backup copy (BC) deallocation) *Let t_i be a task having two task copies PC_i and BC_i. If PC_i was correctly executed, then BC_i can be deallocated and free its slot for new arriving tasks.*

Definition 2 (Backup copy (BC) overloading) *Let P_x be a processor and t_i and t_j be two tasks. Backup copies BC_i and BC_j can overlap each other on the same processor unless $PC_i \in P_x$ and $PC_j \in P_x$ because, if a fault occurs on the processor P_x, both backup copies BC_i and BC_j may need to be executed.*

In our research, we make use of notations summed up in Table 3.1. Using this notations, the conditions for the primary/backup approach are as follows:

Condition 1 (No overlap in time between primary and backup copies of the same task) *Let t_i be a task having two task copies PC_i and BC_i. BC_i cannot start its execution before the end of PC_i, i.e. $end(PC_i) \leqslant start(BC_i)$. Otherwise BC_i needs to be executed (at least during the overlap with PC_i if the backup deallocation is authorised), which causes the system overheads.*

Condition 2 (Respect of real-time constraints) *Let t_i be a task having two task copies PC_i and BC_i and Condition 1 applies. No copies can start before the task arrival and they must be executed prior to the task deadline, i.e. $a_i \leqslant start(PC_i) < end(PC_i) \leqslant start(BC_i) < end(BC_i) \leqslant d_i$. Otherwise the input data may not be available and the results may not be useful anymore.*

Condition 3 (Primary copy and backup copy processor constraint) *Let t_i be a task having two task copies PC_i and BC_i. PC_i and BC_i cannot be scheduled on the same processor P_x, i.e. $PC_i \in P_x \Rightarrow BC_i \notin P_x$. Otherwise, if a fault occurs during the execution of PC_i, the processor P_x may not recover and the execution of BC_i may be faulty too.*

Condition 4 (No overlap in space of primary copies on the same processor) *Let PC_i and PC_j be respectively primary copies of t_i and t_j. A processor P_x can execute only one primary copy at the same time, i.e. (PC_i and PC_j) $\in P_x \Rightarrow end(PC_i) \leqslant start(PC_j)$ or $end(PC_j) \leqslant start(PC_i)$.*

3.1.1.1 Mathematical Programming Formulation

In this section, 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

$$\begin{cases} \textbf{1)}\ PC_i \text{ scheduled } \Leftrightarrow BC_i \text{ scheduled} \\ \textbf{2)}\ a_i \leqslant start(PC_i) < end(PC_i) \leqslant start(BC_i) < end(BC_i) \leqslant d_i \\ \textbf{3)}\ PC_i \in P_x \Rightarrow BC_i \notin P_x \\ \textbf{4)}\ (PC_i \text{ and } PC_j) \in P_x \Rightarrow end(PC_i) \leqslant start(PC_j) \text{ or } end(PC_j) \leqslant start(PC_i) \\ \textbf{5)}\ (BC_i \text{ and } BC_j) \in P_x \Rightarrow end(BC_i) \leqslant start(BC_j) \text{ or } end(BC_j) \leqslant start(BC_i) \end{cases}$$

The purpose of the objective function is maximising the number of accepted tasks, which is equivalent to minimising the task rejection rate. The first two constraints are related to the principle of the PB approach, i.e. every task has two no overlapping copies, which are delimited by the arrival time and deadline. The third constraint forbids the primary and backup copies of the same task to be scheduled on the same processor. The last two constraints account for no overlap among task copies on one processor, i.e. only one task copy can be scheduled per processor at the same time. Whereas the fourth constraint must be respected all the time, the fifth constraint is used only when the BC overloading is not authorised.

3.1.1.2 Processor Allocation Policies

Three processor allocation policies are presented in this thesis: the *exhaustive search*, the *first found solution search processor by processor* and the *first found solution search slot by slot*. Algorithm 6 summarises the main scheduling steps independent of the processor allocation policy.

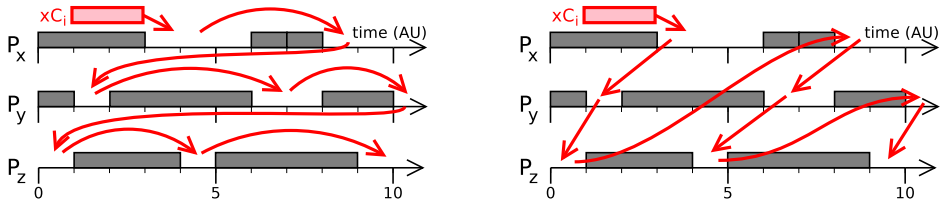
The *exhaustive search* (ES) tests all (P) processors to find a primary copy slot and ($P-1$) processors to search for a backup copy slot in order to respect Condition 3. After such a search, the algorithm provides the best solution, i.e. the one having its primary copy scheduled as soon as possible and the backup copy placed as late as possible.

Algorithm 6 Primary/backup scheduling

Input: Task t_i, Mapping and scheduling MS of already scheduled tasks
Output: Updated mapping and scheduling MS

```
1: if new task t_i arrives then
2:     Map and schedule PC_i
3:     Map and schedule BC_i
4:     if PC and BC slots exist then
5:         Commit the task t_i
6:     else
7:         Reject the task t_i
```

The second and third processor allocation policies are mainly meant for real-time systems, which may not have time to search for a solution on all processors, assess all possibilities and then opt for the best one. The idea is to find a solution as quickly as possible and not necessarily the best one. Naedele [103] presented the *sequential search*, which we call the *first found solution search - processor by processor* (FFSS PbP). The algorithm goes through processors, one by one, until it finds a slot large enough to place a copy or until it scours all processors, as it is depicted in Figure 3.2a. There is no restriction on scheduling, which means that a primary copy can be scheduled rather late within the task window. This decreases the chance to place the corresponding backup copy within the remaining scheduling window and subsequently increases the task rejection rate.



(a) Processor by Processor (PbP) (b) Slot by Slot (SbS)

Figure 3.2 – Principle of the First Found Solution Search (FFSS)

In order to improve the previous method and favour placing primary copies as soon as possible, we propose the processor allocation policy called the *first found solution search - slot by slot* (FFSS SbS). It starts to check the first free slot on each processor and then, if solution is not found, it continues with next slots (second, third, ...) until a solution is obtained or it tests all free slots on all processors. The principle of this policy is illustrated in Figure 3.2b.

The selection of the processor on which the search for a slot starts plays an important role in the system schedulability and workload distribution among processors [103]. Therefore, to avoid a non-uniformity of the processor load for both PbP and SbS, the FFSS for a primary copy slot starts on the processor following the processor on which the primary copy of the previous task was successfully scheduled. The search then continues in increasing order of the processors until a slot is found or all processors are scoured [103]. If a primary copy slot of a new task is found on processor P_x, a search for a backup copy slot is carried out. It starts on processor P_{x-1} and it proceeds in decreasing order of the processors till a slot is found or no more processor is available.

3.1.1.3 Scheduling Search Techniques

There exist several techniques to search for schedules of primary and backup copies. In this manuscript, we analyse two of them: one presented by Ghosh et al. [61], which we call the *free slot search technique* (FSST), and one introduced by Zheng et al. [155] and named the *boundary schedule search technique* (BSST). Since the latter technique is not compatible with one of our objectives, i.e. to reduce the algorithm

run-time, as it will be shown later, it is used only to draw a comparison with the former technique. Therefore, unless otherwise stated (see Section 3.1.3.8), the FSST is considered.

Free Slot Search Technique

When searching for a slot for an arriving task, the FSST compares the length of the current free slot [1] with the task computation time. If the current free slot is large enough, a task copy can be scheduled on it subject to the processor selection policy described in Section 3.1.1.2.

As the primary copies should be placed as soon as possible, the search for a primary copy slot starts at the task arrival time and then continues checking the duration of every free slot within the scheduling window until a solution on a given processor is found or all free slots tested. If a free slot is large enough, a primary copy is placed at its beginning.

The search for a backup copy slot starts at the task deadline in order to find a slot as late as possible. If the BC overloading is not authorised, the algorithm checks free slots as previously. Otherwise, it checks slots delimited by primary copies and non-overloadable backup copies because two backup copies having their respective primary copies on the same processor cannot overload each other. The search thus continues verifying the duration of available slots within the scheduling window up to a slot on a given processor is available or all slots tested. If a slot is large enough, a backup copy is placed at its end.

Figure 3.3 shows two processor schedules. The green solid lines identify the free slots and the red dotted lines the free slots when scheduling a backup copy and the BC overloading is authorised. All backup copies are considered as overloadable.

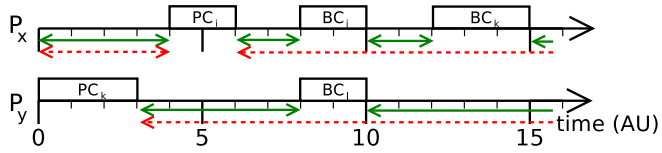


Figure 3.3 – Examples of free slots

Boundary Schedule Search Technique

While the primary copies are always placed as soon as possible, the scheduling of the backup copies using the BSST is not so straightforward. Actually, to maximise the BC overloading (if authorised), the computation of the percentage between overlapping backup copies is carried out. A slot having the highest overlap percentage, which means the lowest replication cost as defined in Section 2.4.4, is chosen. In case of a tie, the slot with the latest start time is selected.

In order not to compute this cost for all slots on each processor and thus to reduce the algorithm run-time, the authors of [155] consider only *boundary schedules*, i.e. slots having their start time and/or finish time at the same time as already scheduled task copies. In general, the primary copy has two boundaries to place a new task copy, while the overloadable backup copy (if the BC overloading is authorised) has four boundaries to do so, as depicted in Figures 3.4. Every possible attempt to schedule a copy starting/ending at a given boundary is illustrated by a violet arrow, which also indicates its direction. The earliest time when a backup copy can start its execution, i.e. when a primary copy finishes its execution, and the task deadline are also considered as boundaries.

(a) Primary copy (b) Backup copy

Figure 3.4 – Different possibilities to place a new task copy when scheduling using the BSST

1. A *free slot* is a slot on a given processor, where no task copy is placed.

Figure 3.5 depicts three possibilities of slots for a backup copy BC_v. The red dash-and-dot rectangle stands for a non-boundary slot, whereas two green dotted rectangles denote the boundary slots. The percentage indicates the proportion of overlapping among overloadable backup copies.

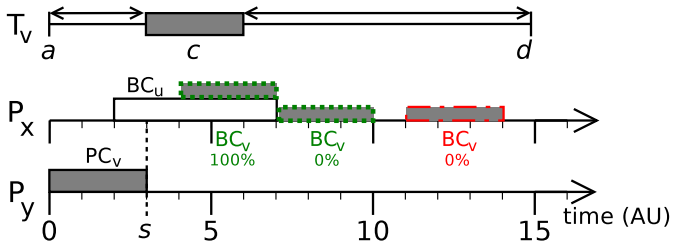


Figure 3.5 – Example of boundary (green) and non-boundary (red) slots

The BSST is primarily meant for the exhaustive search. Nevertheless, we realised several modifications to adapt this scheduling search technique also to the non-exhaustive searches in order to carry out comparisons with other scheduling techniques. These modifications are presented in Appendix A.

3.1.1.4 Active Primary/Backup Approach

Until now, the *passive* primary/backup approach was considered, i.e. the primary and backup copies of the same task cannot overlap each other on two different processors, as stated in Condition 1. Nevertheless, this approach may be too restrictive for some real-time systems since the deadline may be earlier than two times the computation time and, therefore, the *active* primary/backup approach should be considered. This approach was presented in Section 2.4.3.

On the one hand, the active approach allows the primary and backup copies to overlap each other in space and thus facilitates the scheduling of tasks with tight deadlines. On the other hand, it gives rise to the system overheads because the system entirely or partially executes the backup copy (during the execution of the corresponding primary copy). Besides, the active approach adds more schedulability constraints: the backup copies scheduled by means of this method cannot overload other backup copies and cannot be overloaded as they always need to be executed (in total or in part).

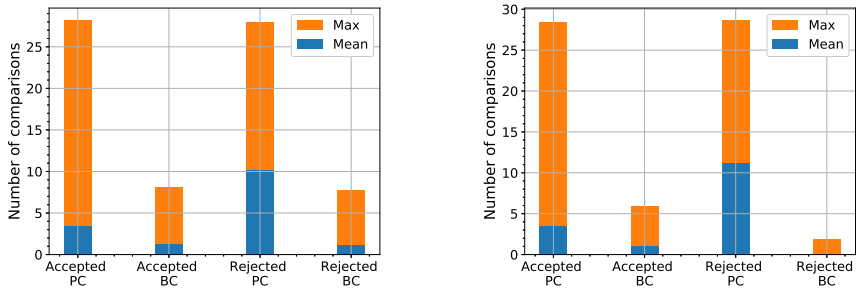
3.1.1.5 Limitation on the Number of Comparisons

When scheduling a task, the simplest idea aiming at reducing the algorithm run-time is to limit the number of comparisons between the free slot duration and the computation time c_i [103]. 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.

To justify this idea, we found out that accepted tasks require less comparisons than rejected tasks (in terms of mean values) and the mean number of comparisons is significantly lower than the maximum number of comparisons, as shown in Figures 3.6. These figures represent the mean and maximum numbers of comparisons per task for the PB approach with BC deallocation with and without BC overloading using the FFSS SbS ($P = 14$, $TPL = 1.0$) [2] without limitation on the number of comparisons. Consequently, when scheduling a new task, the probability that it will be successfully scheduled is lower when the number of comparisons is already high.

The detailed analysis of the numbers of comparisons for accepted and rejected copies showed that the number of comparisons when scheduling a primary copy depends on the number of processors, while the

2. The *Targeted Processor Load* (TPL), defined in Section 3.1.2.1, is a parameter related to the theoretical processor load when generating task arrivals. If $TPL = 1.0$, the arrival times are generated so that every processor is considered to be working all the time at 100%.

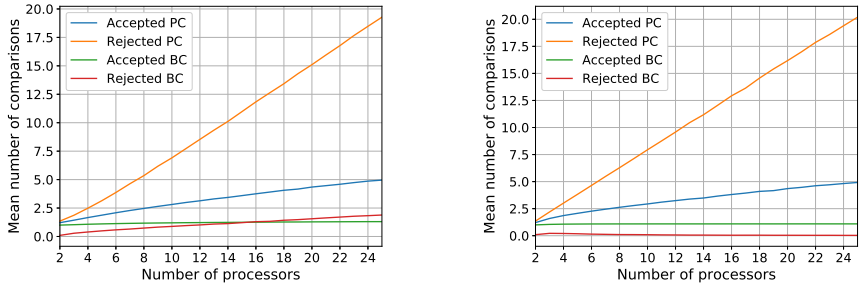


(a) PB approach + BC deallocation

(b) PB approach + BC deallocation + BC overloading

Figure 3.6 – Mean and maximum numbers of comparisons per task (FFSS SbS, $P = 14$, $TPL = 1.0$, no limitation on the number of comparisons)

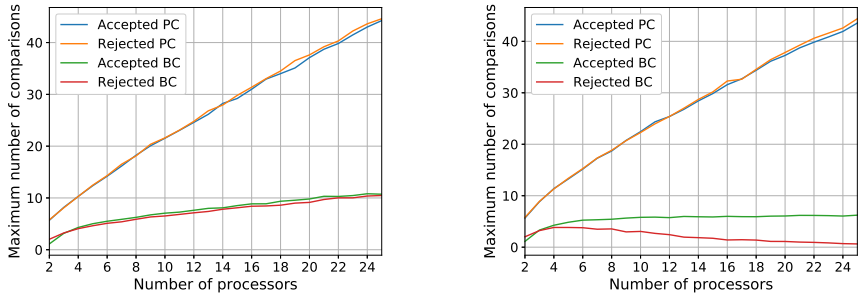
one for a backup copy is almost independent of the number of processors. This is noticeable in Figures 3.7 and 3.8 respectively depicting the mean and maximum numbers of comparisons per task as a function of the number of processors for the PB approach with BC deallocation with and without BC overloading without any limitation on the number of comparisons (FFSS SbS, $TPL = 1.0$).



(a) PB approach + BC deallocation

(b) PB approach + BC deallocation + BC overloading

Figure 3.7 – Mean numbers of comparisons per task as a function of the number of processors (FFSS SbS, $TPL = 1.0$, no limitation on the number of comparisons)



(a) PB approach + BC deallocation

(b) PB approach + BC deallocation + BC overloading

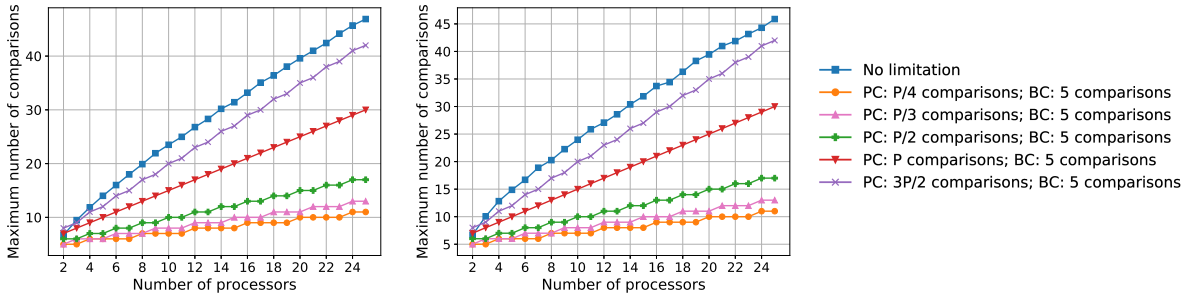
Figure 3.8 – Maximum number of comparisons per task as a function of the number of processors (FFSS SbS, $TPL = 1.0$, no limitation on the number of comparisons)

Regarding the backup copies, the mean number of comparisons is between 1 and 2 and the maximum number of comparisons can exceed 10. In our simulations, we set the BC threshold at 5 to avoid that a task is often rejected due to missing free slot for a backup copy. Therefore, we define the theoretical maximum value of the run-time rt_{limit} as reads:

$$rt_{limit} = rt_{limit}(PC) + rt_{limit}(BC) = \gamma \cdot P + 5 \tag{3.1}$$

where γ is the limitation coefficient for primary copies expressed in our simulation framework as a function of the number of processors.

To illustrate Equation 3.1, Figures 3.9 plot the theoretical limitation on the maximum number of comparisons per task for the PB approach with BC deallocation with and without BC overloading as a function of the number of processors (FFSS SbS, $TPL = 1.0$). As a baseline, represented by the blue curve, we make use of our experimental results when a limitation is not considered.



(a) PB approach + BC deallocation

(b) PB approach + BC deallocation + BC overloading

Figure 3.9 – Theoretical limitation on the maximum number of comparisons per task as a function of the number of processors (FFSS SbS, $TPL = 1.0$)

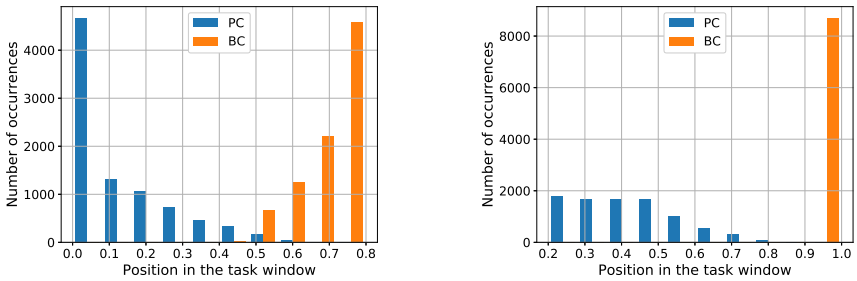
3.1.1.6 Restricted Scheduling Windows

The second method to reduce the algorithm run-time when scheduling a task is called the *restricted scheduling windows*. Before giving a definition, we examine positions of the primary and backup copies within the task window. As an example, we consider a 14-processor system (with $TPL = 1.0$) without using the method of restricted scheduling windows. In such a case, the numbers of occurrences, where the primary and backup copies respectively start or finish their execution, as a function of the position in the task window are depicted in Figures 3.10. The results are shown for the PB approach with BC deallocation but they are almost the same for the PB approach with BC deallocation and with BC overloading.

It can be seen that, although the algorithm tries to schedule the primary copies as soon as possible, a non-negligible amount of them starts later than at the task arrival time, as illustrated in Figure 3.10a. Regarding the backup copies, the majority of them finishes at the task deadline thanks to the BC deallocation, as depicted in Figure 3.10b.

Therefore, the aim of the method of restricted scheduling windows is threefold:

1. to avoid the mutual scheduling interference between primary and backup copies of the same task,
2. to reduce the run-time (measured by means of the number of comparisons carried out before definitely accepting or rejecting a task),
3. to favour placing 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) Start time (b) End time

Figure 3.10 – (a) Number of occurrences of task start time; (b) number of occurrences of task end time as a function of the position in the task window (PB approach + BC deallocation; FFSS SbS; $P = 14$; $TPL = 1.0$)

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 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 3.11.

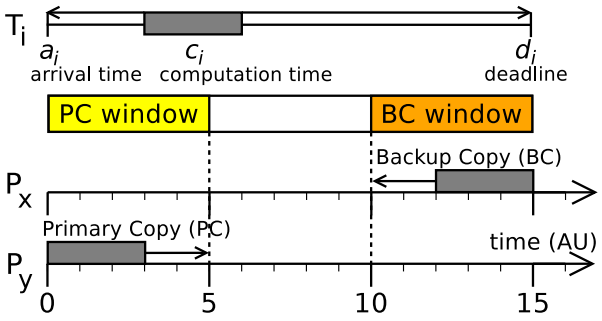


Figure 3.11 – Primary/backup approach with restricted scheduling windows ($f = 1/3$)

To theoretically evaluate the worst-case run-time when placing tasks using the restricted scheduling windows, we were inspired by [155]. We define N_{ps} the number of all possible slots where a copy can be placed. To simplify, we consider that N_{ps} is uniformly distributed within the task window and has the same value on all processors. This number is not easy to estimate in advance and that is why experimental results are essential to observe the trend. The value of N_{ps} is not the same for the primary and backup copies as it can be seen for example in Figures 3.8. Therefore, $N_{ps}(PC)$ and $N_{ps}(BC)$ denote the number of all possible slots within the scheduling window when placing a primary copy or a backup copy, respectively. We remind the reader that α is a multiple of the computation time to define the size of the task window and that a backup copy cannot be scheduled on the same processor as the primary copy. Thereby, the theoretical maximum value of the run-time rt_{RSW} is expressed as follows:

$$
\begin{aligned}
rt_{RSW}(PC) &= P \cdot N_{ps}(PC) \cdot \max\left(\frac{1}{\alpha}; \min\left(1 - \frac{1}{\alpha}; f\right)\right) \\
rt_{RSW}(BC) &= (P-1) \cdot N_{ps}(BC) \cdot \max\left(\frac{1}{\alpha}; \min\left(1 - \frac{1}{\alpha}; f\right)\right) \\
rt_{RSW} &= rt_{RSW}(PC) + rt_{RSW}(BC)
\end{aligned}
\tag{3.2}
$$

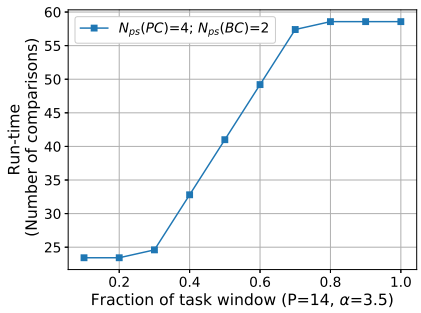


Figure 3.12 – Example of theoretical maximum run-time

Figure 3.12 shows a trend of the theoretical maximum run-time. Since it is a worst-case, we assume that all slots within the PC and BC scheduling window are respectively tested. It can be observed that, when the fraction of task window decreases, the run-time, expressed as the number of comparisons, is reduced because there are less possible slots to test.

3.1.1.7 Several Scheduling Attempts

The previous two devised enhancements of the PB approach mainly dealt with the reduction in the algorithm run-time, whereas the method described in this section focuses on the decrease in the rejection rate. Up to now, the algorithm had only one attempt to schedule a task and it was carried out at the arrival time a_i. However, it may sometimes happen that a task is rejected at the task arrival even though several time units later there is a slot freeing up and large enough to accommodate a task copy thanks to the BC deallocation. The aim of the proposed method is to retry the scheduling later, at the percentage ω of the task window tw_i, and thus to increase the chance for a task to be accepted.

An example for $\omega = 25\%$ is illustrated in Figure 3.13. Algorithm 7 sums up the main scheduling steps of this method.

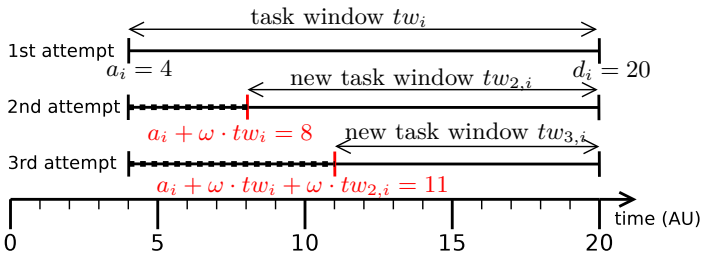


Figure 3.13 – Three scheduling attempts at $\omega = 25\%$

Algorithm 7 Algorithm using the method of several scheduling attempts

Input: Task t_i, Mapping and scheduling MS of already scheduled tasks
Output: Updated mapping and scheduling MS

```
1: if new task t_i arrives then
2:     Search for PC and BC slots for the first time
3:     if PC and BC slots exist then
4:         Commit the task t_i
5:     else
6:         while task not scheduled and new attempt authorised do
7:             Compute the time of new attempt to schedule the task
8:             Search for PC and BC slots
9:             if PC and BC slots exist then
10:                Commit the task t_i
11:        if task not scheduled and new attempt not authorised then
12:            Reject the task t_i
```

To evaluate the efficiency of this method, we define the *percentage of slack within the task window* ps_i of task t_i as follows:

$$ps_i = \frac{s_i}{tw_i} = \frac{tw_i - 2 \cdot c_i}{tw_i} \tag{3.3}$$

where s_i denotes the *slack*, i.e. the remaining time within the task window tw_i after subtracting twice the computation time c_i necessary for the primary and backup copies to be executed. The higher the percentage ps_i, the higher the chance to schedule a task later than at its arrival time. Nonetheless, the higher the number of scheduling attempts for one task, the lower the probability for new scheduling attempts to be successful because there is less and less slack.

3.1.2 Experimental Framework

In this section, we describe our simulation scenario and define metrics used to evaluate our algorithms.

3.1.2.1 Simulation Scenario

Table 3.2 sums up the simulation parameters. For each simulation scenario, 100 simulations of 10 000 tasks were treated and the obtained values were averaged. Unless the simulations with fault injections are carried out (see Section 3.1.3.13), we consider that no fault occurs during our simulations. Therefore, if the BC deallocation is put into practice, all backup copies are deallocated when their respective primary copies finish.

The arrival times are generated using the Poisson distribution with parameter λ expressed as follows:

$$\lambda = \frac{average\ c}{TPL \cdot P} \tag{3.4}$$

depending on the computation time, number of processors and targeted processor load. If the *Targeted Processor Load* (TPL) equals 1.0, the arrival times are generated so that every processor is considered to be working all the time at 100%.

To compare our results, we defined the mathematical programming formulation of our problem as described in Section 3.1.1.1 and carried out resolutions in CPLEX optimiser using the same data set.

The problem is solved in CPLEX optimiser [3], which is a high-performance mathematical programming solver for linear programming, mixed-integer programming and quadratic programming. Since tasks aperiodically arrive and backup copies are deallocated once their corresponding primary copies are correctly

3. `https://www.ibm.com/analytics/cplex-optimizer`

Table 3.2 – Simulation parameters

Parameter	**Distribution**	**Value(s) in** ms
Number of processors P		2 – 25
Computation time c	Uniform	1 – 20
Arrival time a	Poisson	$\lambda = \frac{average\ c}{TPL \cdot P}$
Deadline d	Uniform	$[\![a + 2c; a + 5c]\!]$

executed, a dynamic aspect needs to be modelled in CPLEX solver. It means that it is not possible to resolve the scheduling problem only once because CPLEX optimiser would know all task characteristics in advance and it would be an offline instead of an online scheduling.

Therefore, it is necessary to update the task data set in the course of time and to carry out a new resolution when a new task arrives. We make use of the main function managing this dynamic aspect. Its main steps are encapsulated in Algorithm 8. At each task arrival, the main function updates task data: new task arrivals (Line 3) and deallocated backup copies (Line 4); launches a new resolution using the current data set (Line 5) and removes rejected task from the current data set (Line 6). After the last task arrival, it deallocates the remaining backup copies (Line 7) and computes the performances of the optimal solution (Line 8).

Algorithm 8 Main steps to find the optimal solution of a scheduling problem in CPLEX optimiser

Input: Task data set
Output: Mapping and scheduling of the optimal solution
1: Initialise the current data set and model
2: **for** each time when a task arrives **do**
3: Add a new task to the task set
4: Remove all backup copies which can be deallocated from the task set
5: Solve the problem
6: Remove all unscheduled tasks from the task set
7: Remove all backup copies which can be deallocated from the task set
8: Compute the rejection rate and processor load of the optimal solution

Due to computational time constraints, only 16 resolutions using CPLEX optimiser were conducted and the results were averaged. To illustrate such constraints, if we sum the time elapsed to find an optimal schedule for systems with processors respectively ranging from 2 to 25, one simulation took on the average of 16 simulations 72.61 hours (the maximum duration is 98.05 hours, while the minimal one is 48.97 hours) when 12 server processors were used. More details on CPLEX parameters are described in Appendix C.

Fault Generation

Before explaining how simulations with faults are conducted, we focus on the fault generation. We were inspired by the two state discrete Markov model of the Gilbert-Elliott type, which was described in Section 1.3.2. Since we assume a rather short simulation duration and a harsh environment, we simplified this model to only one state, which is considered as "bursty".

When we generate faults at the task level to carry out simulations with fault injections, we make use of the Python function `random`. This function generates a random float within the interval $[0; 1)$. To implement this function, Python uses the *Mersenne Twister* as the core generator, which produces 53-bit precision floats and has a period of $2^{19937} - 1$ [120].

Since we consider that faults are independent, we generate a random number at each time step ($1\,ms$ in our simulations) for each processor. This generated number is then compared to the fault rate (mostly between $1 \cdot 10^{-6}$ and $1 \cdot 10^{-1}$ fault per ms). If it is smaller than the threshold defined by the fault rate, a fault is generated. Otherwise, there is no generated fault.

For simulations with faults, we take into account that the estimated processor fault rate is $1/120h^{-1} = 2.3 \cdot 10^{-6}$ fault/s [47], which corresponds to $5.8 \cdot 10^{-5}$ fault/s for 25-processor system [4]. Therefore, we randomly inject faults at the level of task copies with fault rate for each processor between $1 \cdot 10^{-5}$ and $5 \cdot 10^{-2}$ fault/ms in order to assess the algorithm performances not only in real conditions but also in a harsher environment. Consequently, the assumption about only one processor failure at the same time may not be respected for higher fault rates [5], which may cause that a task having both primary and backup copies impacted does not contribute to the system throughput, defined in Section 3.1.2.2. For the sake of simplicity, we consider only transient faults and one fault can impact at most one task copy.

3.1.2.2 Metrics

The evaluation of the algorithm performances was based on the following metrics.

The *rejection rate* is defined as the ratio of rejected tasks to all arriving tasks to the system. The *system throughput* 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 *ratio of computation times* is the proportion of the sum of the computation times of accepted tasks to the sum of the computation times of all arriving tasks to the system. The *processor load* stands for the effective system load taking into account the BC deallocation and rejection rate.

The *percentage of backup copies in rejected tasks* is defined as the proportion of backup copies in all rejected tasks.

To evaluate the system resiliency, we make use of the *Time To Next Fault* (TTNF) [61], which is the time elapsed between a chosen time instant and the time when a new fault may occur not violating the assumption about only one fault in the system at the same time. It is expressed in ms. The lower value, the better. This metric is computed after each successful scheduling of primary copy PC_i considering that a fault occurs at the beginning of PC_i.

The algorithm run-time is evaluated by the *number of comparisons* accounting for the number of tested slots. One comparison accounts for one evaluation whether a slot is large enough to accommodate a task copy (PC or BC) on a given processor. All tasks are taken into account, no matter whether they are finally accepted or rejected. This metric is essential for embedded systems because it is related to the energy consumption and rate of scheduling.

As our algorithm is meant for embedded systems dealing with hard real-time tasks, we try to reduce the algorithm run-time as much as possible without worsening system performances. Therefore, our aim is to first and foremost cut down on the number of comparisons and then decrease the rejection rate.

3.1.3 Results

This section presents results of various techniques introduced for the PB approach in this chapter. We first analyse the baseline results with and without the BC deallocation and study whether or not the algorithm is biased when it rejects tasks. Then, we evaluate the active PB approach and different processor allocation policies and scheduling searches. Next, we present the results showing the overheads of the primary/backup approach and the comparison with the optimal solution provided by CPLEX solver. Later on, we separately analyse three enhancing methods (limitation on the number of comparisons, restricted scheduling windows and several scheduling attempts) in order to determine their parameters satisfying the best our objective, i.e. to reduce the algorithm run-time without deteriorating the system

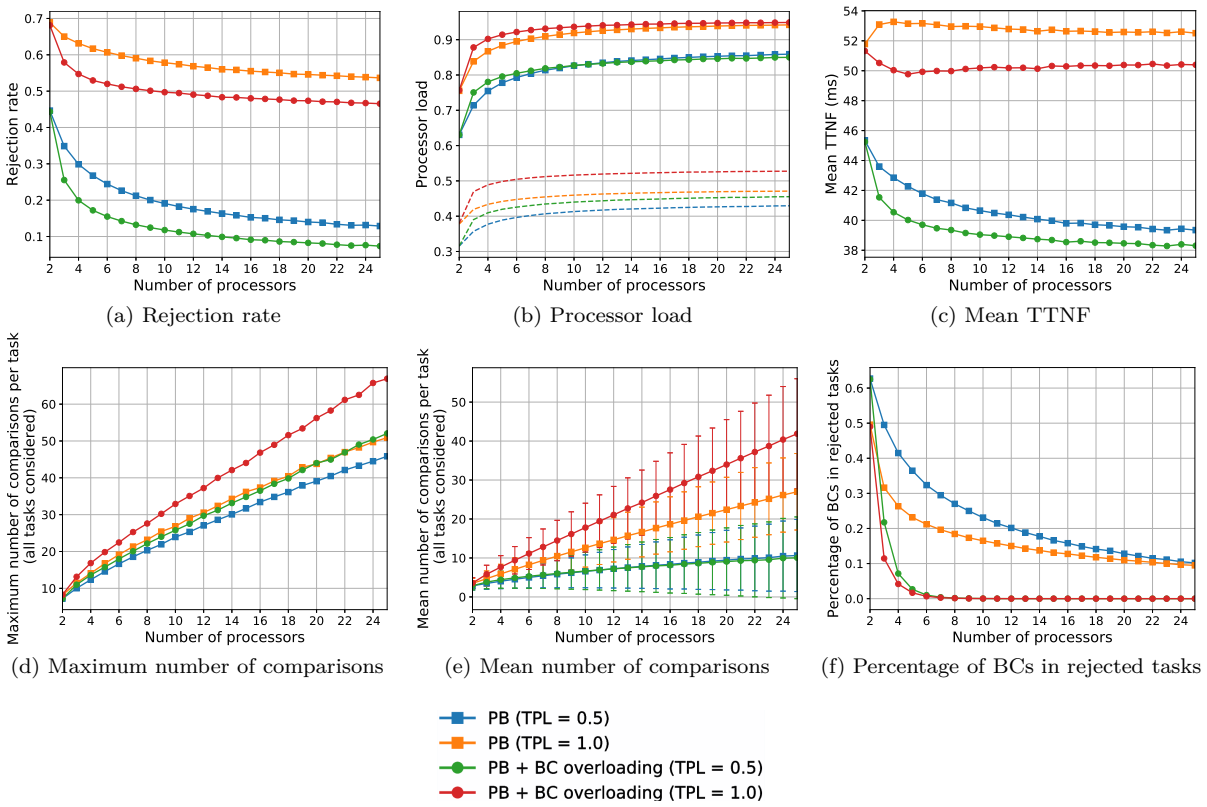
4. We remind the reader that the system reliability is lower than the reliability of its processors, as it is defined by Formula 1.7 introduced in Section 1.3.4.

5. Inspired by [61], we make use of a metric, which we called the *Time To Next Fault* (TTNF) and defined in Section 3.1.2.2. The worst-case value is obtained if a fault occurs at the beginning of the primary copy having the longest c and the largest tw. Consequently, in our scenario $TTNF_{\text{worst-case}} = c_{max} \cdot tw_{max} = 20 \cdot 5 = 100\,ms$, which implies the fault rate of $1 \cdot 10^{-2}$ fault/ms for 25-processor system, i.e. the fault rate of $4 \cdot 10^{-4}$ fault/ms for one processor. Nevertheless, our results (represented in Figure 3.15c depicting the mean TTNF as a function of the number of processors for the PB approach with BC deallocation and with or without BC overloading (FFSS SbS; $TPL = 1.0$)) show that the mean value of TTNF is less than one half of the worst-case TTNF no matter the chosen method.

performances. These methods are then combined together and their performances are assessed. Finally, we evaluate the fault tolerance of the PB approach using the best choice of enhancing techniques.

3.1.3.1 Baseline Results

To analyse the system performances, we study the following metrics: the rejection rate, the processor load, the mean TTNF and the maximum and mean numbers of comparisons per task, and the percentage of backup copies in rejected tasks. Figures 3.14 represent these studied metrics as a function of the number of processors for the PB approach with and without BC overloading. The targeted processor load equals 0.5 and 1.0, respectively, and the chosen processor allocation policy is the FFSS SbS because it will be demonstrated in Section 3.1.3.5 that this policy achieves the lowest rejection rate with a reasonable number of comparisons.



(a) Rejection rate (b) Processor load (c) Mean TTNF

(d) Maximum number of comparisons (e) Mean number of comparisons (f) Percentage of BCs in rejected tasks

Figure 3.14 – System metrics as a function of the number of processors and TPL (PB approach with and without BC overloading; FFSS SbS)

First of all, we note that the results of the rejection rate for the PB approach with BC overloading are better than the ones for the PB approach alone (for example 14% for a 14-processor system with $TPL = 1.0$). In fact, the implemented technique allows the backup copies to overload each other, unless their primary copies are scheduled on the same processor, which saves up free slots that can be used for new arriving tasks. Regarding the processor load, both approaches reach similar values.

It can be seen in Figure 3.14a that the rejection rate decreases with increasing number of processors. We remind the reader that the targeted processor load is set as a constant. Thus, according to the definition of the Poisson distribution parameter λ in Equation 3.4, when the number of processors increases, the parameter λ decreases, which implies that tasks have shorter interarrival time and arrive more often. Actually, the addition of processors brings more possibilities to find a suitable slot so that the processor

load eventually decreases. Besides, the higher the targeted processor load, the higher the rejection rate because the system becomes more charged and consequently rejects more tasks.

Regarding the percentage of the backup copies in the rejected tasks depicted in Figure 3.14f, this percentage decreases when the number of processors increases. The PB approach using the BC overloading has lower percentage of BCs in the rejected tasks than the PB approach, which does not take advantage of this technique.

The processor load is plotted in Figure 3.14b. The solid lines account for the workload of the whole system, i.e. when the primary and backup copies are taken into account. The processor load increases with the system consists of more processors. It can be noticed that, since the BC deallocation is not used, the system performs the same computation twice even though only one execution is necessary in a fault-free environment. The dashed lines in Figure 3.14b stand for the processor load when the backup copies are not considered. Actually, it is the effective processor load from the user's point of view. Even when the BC overloading is put into practice, the effective processor load is about 50%. Consequently, to improve the system performances, the BC deallocation should be introduced and analysed, which is the aim of Section 3.1.3.2.

Furthermore, it can be seen in Figures 3.14a and 3.14b that the rejection rate or processor load as a function of the number of processors do not considerably vary when the number of processors is greater than 12. Therefore, a 14-processor system will be taken as a standard for our comparative computations throughout this manuscript when we illustrate a phenomenon for a given number of processors.

Figure 3.14c represents the mean time to the next fault accounting for the system resiliency. While this metric slightly decreases when the number of processors increases for $TPL = 0.5$, it remains almost constant for $TPL = 1.0$.

The maximum and mean numbers of comparisons per task are shown in Figures 3.14d and 3.14e, respectively. The more processors are in the system, the more comparisons are required. Since the PB approach with BC overloading needs to carry out more comparisons, its number is in general higher. We note that the mean number of comparisons per task is much lower than the maximum one, as it was shown in Figures 3.6.

In addition, Figure 3.14e also depicts the standard deviations for each value of the mean number of comparisons per task. When there are more processors in the system, the mean number of comparisons increases and the standard deviation gets larger as well. When the value of TPL raises, the standard deviation grows because the processor load is higher and there are more comparisons to be carried out and consequently higher chance to have larger standard deviation. We note that the standard deviation is greater for the PB approach with BC overloading than for the PB approach alone (for example for the 14-processor system, the values are respectively 8.4 and 6.4 comparisons per task).

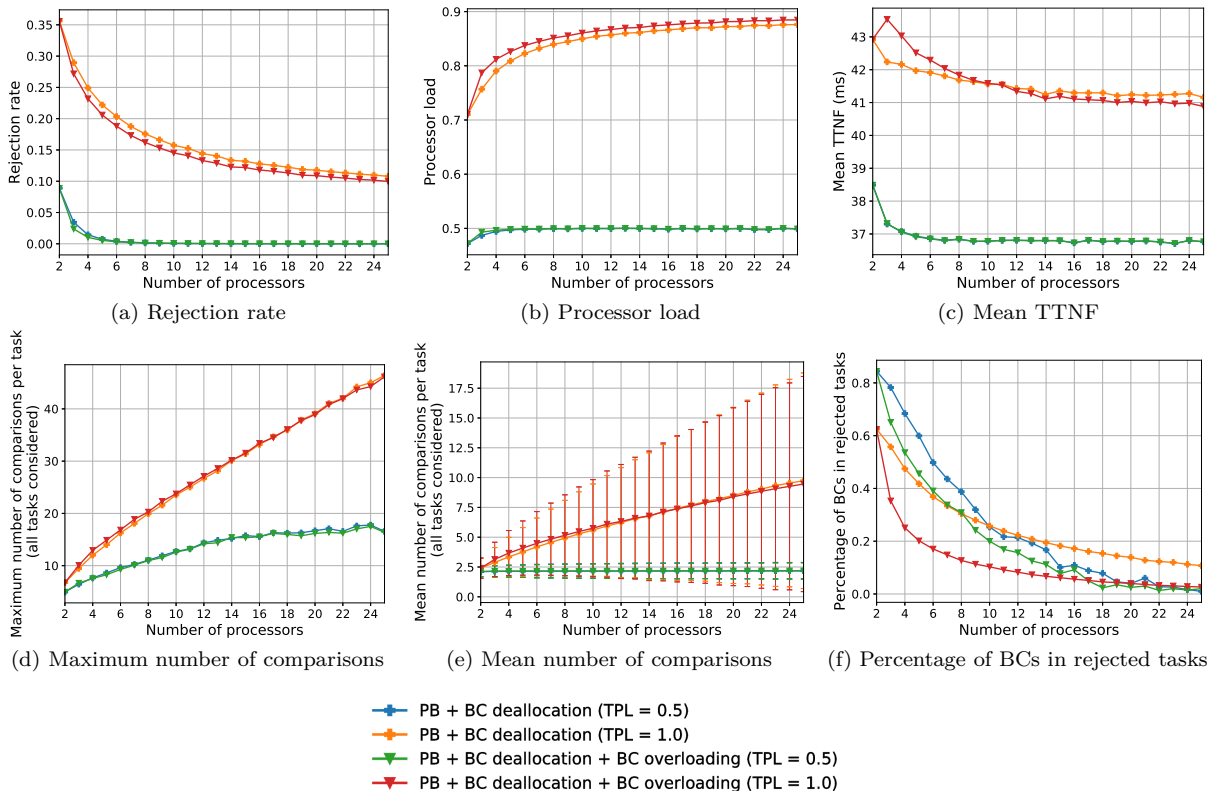
3.1.3.2 Merit of the BC Deallocation

To obtain comparable results to the previous ones, the algorithm makes use of the FFSS SbS and the TPL is fixed at 0.5 and 1.0. Figures 3.15 represent the studied metrics as a function of the number of processors for the PB approach with BC deallocation and with and without BC overloading.

Foremost, it can be noticed that the PB approach with BC deallocation and BC overloading achieves slightly better results than the PB approach with BC deallocation only, which means that the BC deallocation and the BC overloading can be used fruitfully together.

Figure 3.15a shows that the rejection rate is significantly reduced when compared to Figure 3.14a thanks to the BC deallocation. For example for the 14-processor system and $TPL = 1.0$, the gain is about 75% no matter whether the BC overloading is implemented or not.

In addition, Figure 3.15f depicting the percentage of backup copies in rejected tasks demonstrates that a task is generally rejected mainly due to a missing slot for a primary copy. The higher the number of processors, the lower the percentage of the backup copies in the rejected tasks. While the values are slightly greater than 10% for the PB approach with BC deallocation, they are almost 0% for the PB approach with BC deallocation and BC overloading. This leads us to conclude that the BC overloading



(a) Rejection rate (b) Processor load (c) Mean TTNF

(d) Maximum number of comparisons (e) Mean number of comparisons (f) Percentage of BCs in rejected tasks

Figure 3.15 – System metrics as a function of the number of processors and TPL (PB approach with BC deallocation and with and without BC overloading; FFSS SbS)

improves the system schedulability.

The curves of processor load depicted in Figure 3.15b account for both the workload of the whole system and the workload of the primary copies only because all backup copies are deallocated due to no fault occurrence. It means that when the BC deallocation is put into practice, the system can accept twice as more tasks compared to the system without this technique. When a system has higher number of processors, curves tend to the value of the targeted processor load (0.5 or 1.0) showing the effectiveness of the BC deallocation.

Furthermore, thanks to the BC deallocation, the values of the TTNF are lower, which means that a next fault can occur earlier. As Figure 3.15c represents, the values of the mean TTNF for $TPL = 1.0$ are close to 40 AU, which is slightly greater than the average theoretical value computed as $c_{avg} \cdot tw_{avg} = 10.5 \cdot 3.5 = 36.75\,ms$, which is due to the fact that the backup copies are scheduled and deallocated after the correct execution of the corresponding primary copies.

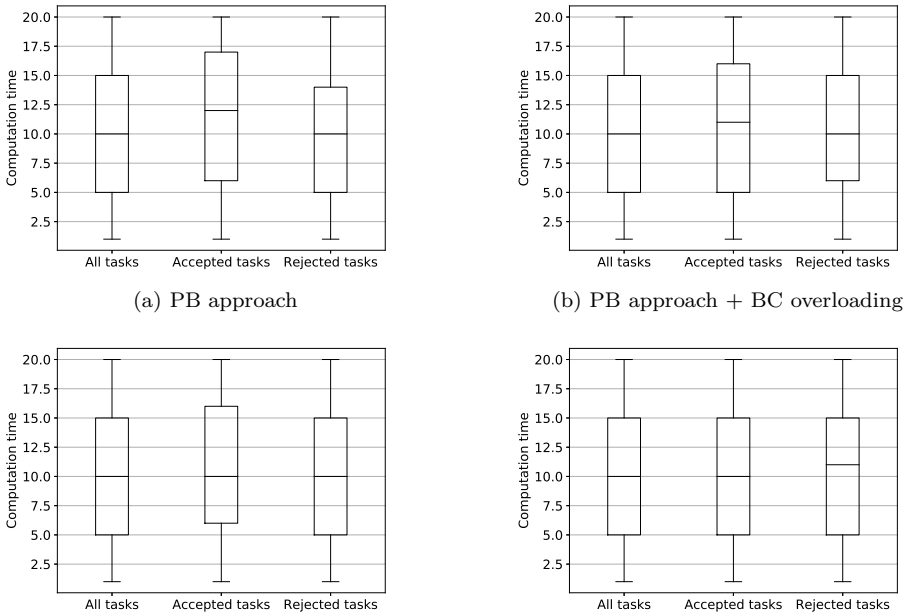
The mean number of comparisons per task and its standard deviations are depicted in Figure 3.15e. While they are almost constant for $TPL = 0.5$, they get larger when the number of processors increases. In fact, when the system is not fully loaded, it is not necessary to carry out a lot of comparisons. When the BC deallocation is put into practice, the value of the standard deviation is independent of the use of the BC overloading. For instance for the 14-processor system, the standard deviation of the PB approach with BC deallocation is 5.3 comparisons per task and the standard deviation of the PB approach with BC deallocation and BC overloading is 5.4 comparisons per task.

The maximum number of comparisons shown in Figure 3.15d increases with the number of processors. Nevertheless, since backup copies are deallocated, there are less comparisons. When $TPL = 1.0$, the maximum number of comparisons is approximately four times higher than the mean one.

3.1.3.3 Bias of Task Rejection Algorithm

An algorithm can be biased if it more likely rejects for example the tasks with shorter computation times. In this section, we evaluate whether the studied algorithm is unbiased in terms of the task rejection. We focus on all arriving, accepted and rejected tasks and compare their statistical distributions with regard to the task computation time. The analysis is carried out by means of the box plots, which are described in Appendix D. The results for a 14-processor system with $TPL = 1.0$ are shown in Figure 3.16.

First of all, it can be noticed that the statistical distribution of all arriving tasks is correctly represented by the simulation parameters summarised in Table 3.2. Although the distribution of accepted and rejected tasks slightly vary for the chosen approach, their distributions remain rather close to the one of arriving tasks. The largest difference is recorded for the mean value of accepted tasks for the baseline PB approach ($11.24ms$) and for the one for the PB approach with BC overloading ($10.72ms$). This allows us to conclude that the studied algorithm has a unbiased behaviour in terms of task rejection.



(a) PB approach

(b) PB approach + BC overloading

(c) PB approach + BC deallocation

(d) PB approach + BC deallocation + BC overloading

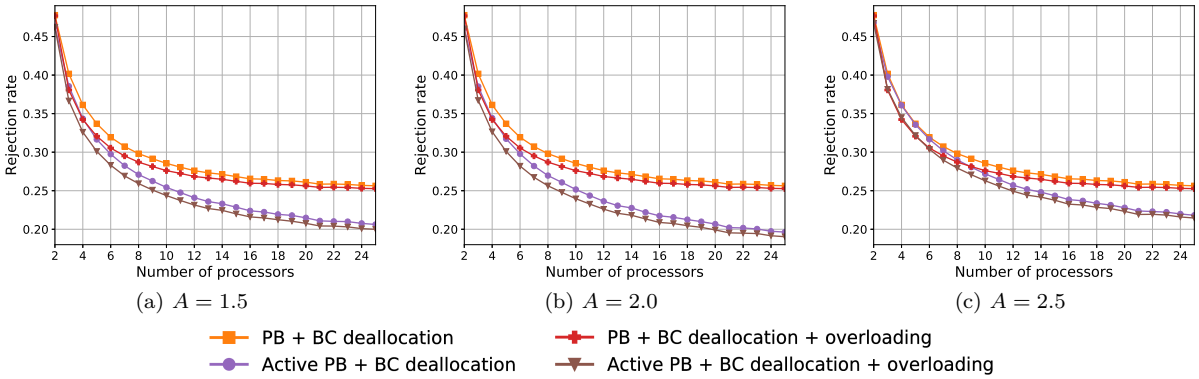
Figure 3.16 – Statistical distribution of tasks with regard to their computation times (FFSS SbS; $TPL = 1.0$; $P = 14$)

3.1.3.4 Evaluation of the Active Primary/Backup Approach

To evaluate the merit of the active PB approach, we make use of the standard simulation parameters as summarised in Table 3.2 but, instead of the size of the task window between $2c$ and $5c$, we consider its size between c and $5c$. Since the passive PB approach requires the size of the task window at least $2c$, this scenario allows us to assess the active PB approach.

The algorithm is based on the FFSS SbS and the TPL is fixed at 1.0. Figures 3.17 depicts the rejection rate as a function of the number of processors for the PB approach with BC deallocation and with or without BC overloading and for their respective versions using the active PB approach.

As it was mentioned in Section 3.1.1.4, the active PB approach induces the system overheads. Consequently, when we employ this approach, we limit its application for tasks with tight deadline. We therefore



(a) $A = 1.5$ (b) $A = 2.0$ (c) $A = 2.5$

Figure 3.17 – Rejection rate of the active and passive PB approach with BC deallocation and with or without BC overloading as a function of the number of processors for different values of the threshold A (FFSS SbS; $TPL = 1.0$)

introduce a threshold A defined as $tw = d - a < A \cdot c$, which determines whether the active approach is used or not. A in Figures 3.17 takes on three values 0.5, 2.0 and 2.5.

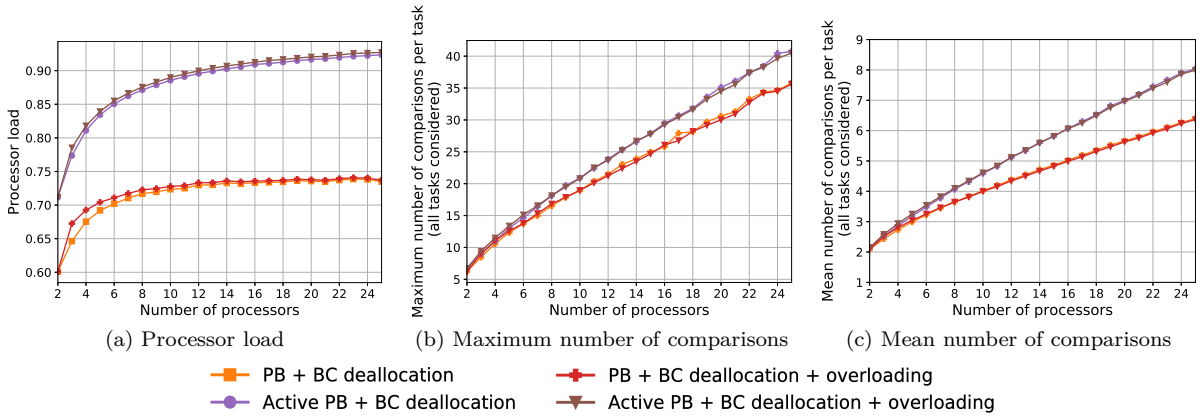
Since the simulation scenario remains the same and only the value of A changes, the results for the passive PB approaches are identical. In general, it can be observed that the active PB approach facilitates the reduction in the rejection rate regardless of the value of A. The lowest rejection rate is obtained when $A = 2.0$ because this value is located at the transition from the passive PB approach to the active one. For example, the active approach for a 14-processor system with $TPL = 1.0$ reduces the rejection rate by 16% for the PB approach with BC deallocation without BC overloading and by 18% for the PB approach with BC deallocation with BC overloading.

In addition, when A is less than 2.0, some tasks are automatically rejected due to the tight deadline, which is the reason why the rejection rate for $A = 1.5$ (Figure 3.17a) is higher than the one for $A = 2.0$ (Figure 3.17b). When A is larger than 2.0, no task is automatically rejected but the merit of the active approach diminishes. In general, the results (not all of them depicted) show that the higher the value of A, the higher the rejection rate of the active approach and therefore the smaller the difference between the passive and active approaches.

Figures 3.18 compare the processor load and the maximum and mean numbers of comparisons per task for the active and passive PB approach with BC deallocation and with or without BC overloading as a function of the number of processors (FFSS SbS, $TPL = 1.0$). The parameter A is set at 2.0 because the active approach using this threshold achieves the lowest rejection rate. We remind the reader that the simulation parameters have changed and subsequently the results are not exactly the same as in the preceding sections.

Figure 3.18a, depicting the processor load, shows that the active approach has always higher processor load than the passive approach. While this phenomenon can be partly explained by lower rejection rate when a system has higher number of processors (26% rejected tasks for the passive approach and 20% rejected tasks for the active approach for a 20-processor system), when the system has only a few processors, both approaches have almost the same rejection rate but the processor load of the active approach is higher (for instance by 19% for a 14-processor system) compared to the passive approach. This shows the non-negligible system overheads of the active approach.

Regarding the maximum and mean numbers of comparisons per task, they are represented in Figures 3.18b and 3.18c, respectively. It can be seen that these numbers are higher for the active rather than for the passive approach and the gap between two approaches gets larger, which again demonstrates the system overheads of the active approach.

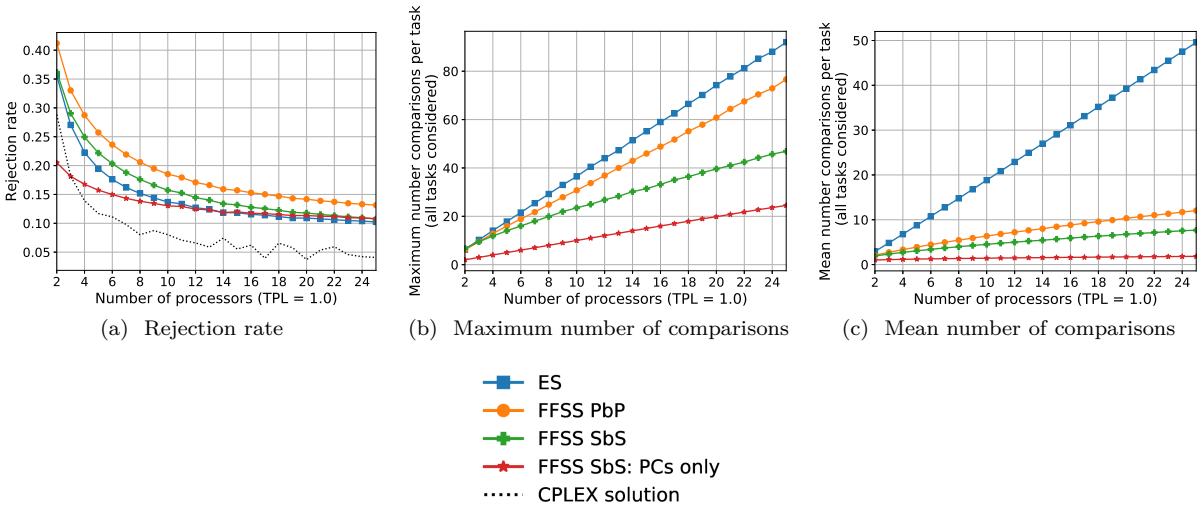


(a) Processor load (b) Maximum number of comparisons (c) Mean number of comparisons

Figure 3.18 – Studied metrics of the active and passive PB approach with BC deallocation and with or without BC overloading as a function of the number of processors (FFSS SbS; $A = 2.0$; $TPL = 1.0$)

3.1.3.5 Comparison of Different Processor Allocation Policies

One of our achievements is a new processor allocation policy called the *first found solution search - slot by slot*. In this section, it is compared to two already existing policies: the *exhaustive search* [61] and the *first found solution search - processor by processor* [103]. The results of the rejection rate, the maximum and mean numbers of comparisons for the PB approach with BC deallocation as a function of the number of processors are depicted in Figures 3.19.



(a) Rejection rate (b) Maximum number of comparisons (c) Mean number of comparisons

Figure 3.19 – Comparison of three processor allocation policies and evaluation of system overheads (PB approach + BC deallocation; $TPL = 1.0$)

Figure 3.19a representing the rejection rate shows that the FFSS SbS achieves better results (by 16% for a 14-processor system) than the FFSS PbP. The rejection rate of the ES is reduced by 11% compared to the FFSS SbS and by 25% compared to the FFSS PbP (both values are related to the 14-processor system). The ES is the best in terms of the rejection rate because this search tests all possible slots and chooses the solution placing the primary copy as soon as possible and the backup copy as late as possible, which contributes to higher schedulability. Nevertheless, it can be seen that the gap between the FFSS SbS and the ES becomes smaller when the number of processors augments.

Regarding the maximum number of comparisons per task plotted in Figure 3.19b, the FFSS SbS reaches notably lower values compared to the ES (for example 41% for 14-processor system) and the FFSS PbP (for instance 29% for the 14-processor system). When considering the mean number of comparisons per task shown in Figure 3.19c, the FFSS SbS requires significantly less comparisons than the ES (for instance reduction by 80% for the 14-processor system) and than the FFSS PbP (for example decrease by 32% for the 14-processor system).

Similar results were obtained for the PB approach with BC deallocation and BC overloading.

Since the FFSS SbS generally performs well, this processor allocation policy is chosen for further experiments.

3.1.3.6 Overhead of the Fault Tolerant Systems

This section assesses the system overheads induced by the PB approach. Figures 3.19 not only plot results of scheduling based on several processor allocation policies for the PB approach with BC deallocation but also the results of scheduling of only primary copies (using the FFSS SbS). The latter results account for a system, which is not fault tolerant.

Even if the BC deallocation is performed when a primary copy finishes, the fault tolerant systems based on the PB approach and having only a few processors have higher rejection rate and higher number of comparisons compared to the systems not providing the fault tolerance. The ES of the fault tolerant systems achieves slightly better results in terms of the rejection rate than the scheduling of only primary copies because the latter makes use of the FFSS SbS, which chooses the first found solution and not necessarily the earliest one. The more processors, the wider the gap in the number of comparisons (since there are more possibilities to test for the PB approach) and the narrower the gap in the rejection rate.

3.1.3.7 Comparison with the Optimal Solution from CPLEX Solver

We compare our proposed processor allocation policy (FFSS SbS) in terms of the rejection rate to the optimal results provided by CPLEX solver, which explored all possible solutions and chose the one minimising the number of rejected tasks. The mathematical programming formulation was given in Section 3.1.1.1.

Figure 3.19a shows that the rejection rate of the FFSS SbS is higher about 5% than the optimal solution and that the algorithm using the FFSS SbS is 2-competitive. This represents a good result taking into account that the proposed technique chooses the first found solution.

The explanation of the difference between the optimal solution from CPLEX solver and the ES is as follows. At time t, the algorithm using the ES deallocates backup copies (if possible) and schedules new tasks one by one. The ES tests all processors for a current task in order to provide a solution, where the primary copy is scheduled as soon as possible and the backup copy is placed as late as possible. By contrast, the CPLEX solver tests all schedules at the same time knowing all tasks available at time t, i.e. new task arrivals and the backup copies, which can be deallocated. It means that primary copies are not necessarily scheduled as soon as possible and backup ones as late as possible.

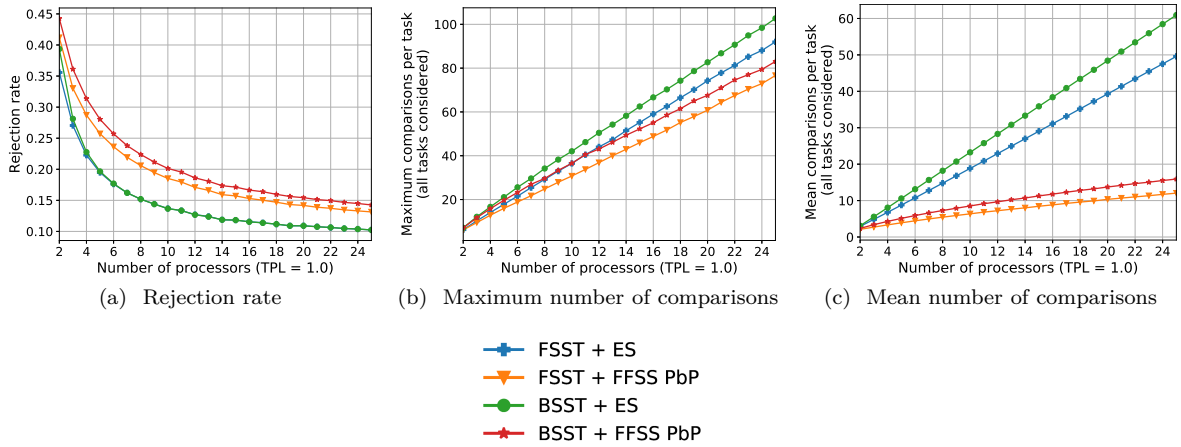
3.1.3.8 Comparison of Scheduling Search Techniques

The aim of this section is to compare two scheduling search techniques presented in Section 3.1.1.3: the *free slot search technique* (FSST) and *boundary schedules search technique* (BSST). Figures 3.20 and 3.21 show the rejection rate, the maximum and mean numbers of comparisons per task for the PB approach with BC deallocation and with or without BC deallocation as a function of the number of processors. In these figures, four curves represent, respectively:

— Free Slot Search Technique + Exhaustive Search (FSST + ES)
— Free Slot Search Technique + First Found Solution Search: Processor by Processor (FSST + FFSS PbP)
— Boundary Schedule Search Technique + Exhaustive Search (BSST + ES)

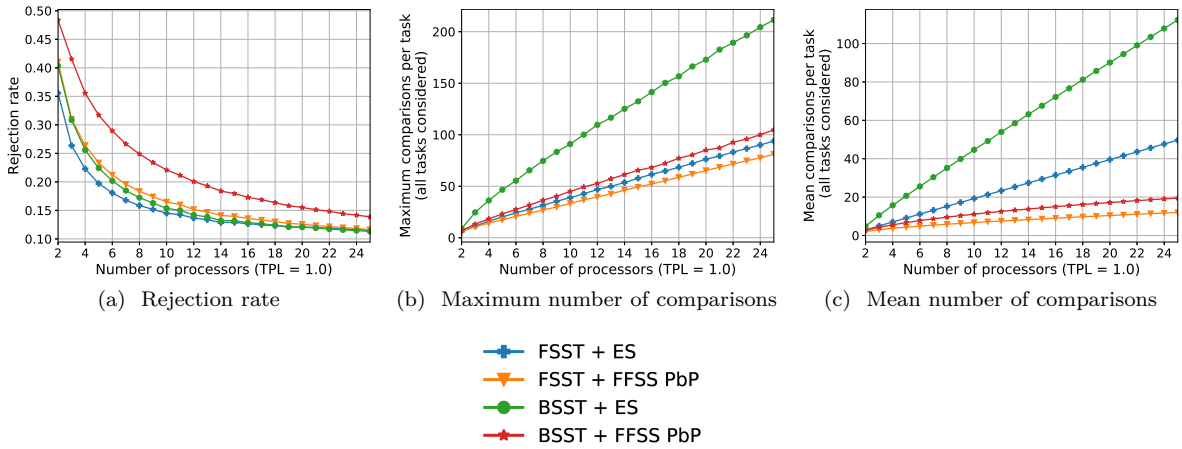
— Boundary Schedule Search Technique + First Found Solution Search: Processor by Processor (BSST + FFSS PbP)

The FFSS SbS is not put into practice for the BSST because it requires even more complex rules than the FFSS PbP described in Appendix A.



(a) Rejection rate (b) Maximum number of comparisons (c) Mean number of comparisons

Figure 3.20 – Comparison of scheduling search techniques (PB approach + BC deallocation; $TPL = 1.0$)



(a) Rejection rate (b) Maximum number of comparisons (c) Mean number of comparisons

Figure 3.21 – Comparison of scheduling search techniques (PB approach + BC deallocation + BC overloading; $TPL = 1.0$)

As it was already demonstrated in Section 3.1.3.5, the ES (independent of scheduling search technique) has lower rejection rate than the FFSS PbP because it scours all processors to choose the solution having the primary copy as soon as possible and the backup copy as late as possible or maximising the overlap with other overloadable backup copies. Nonetheless, this performance is at the expense of higher number of comparisons and thereby longer algorithm run-time.

While the ES for the FSST and the BSST achieves almost the same values of the rejection rate for both of them, the PB approach with BC deallocation and with or without BC overloading, the FSST + FFSS PbP rejects less tasks than the BSST + FFSS PbP. This can be caused by the fact that the principle of "boundary schedules" is not well adapted for a non-exhaustive search.

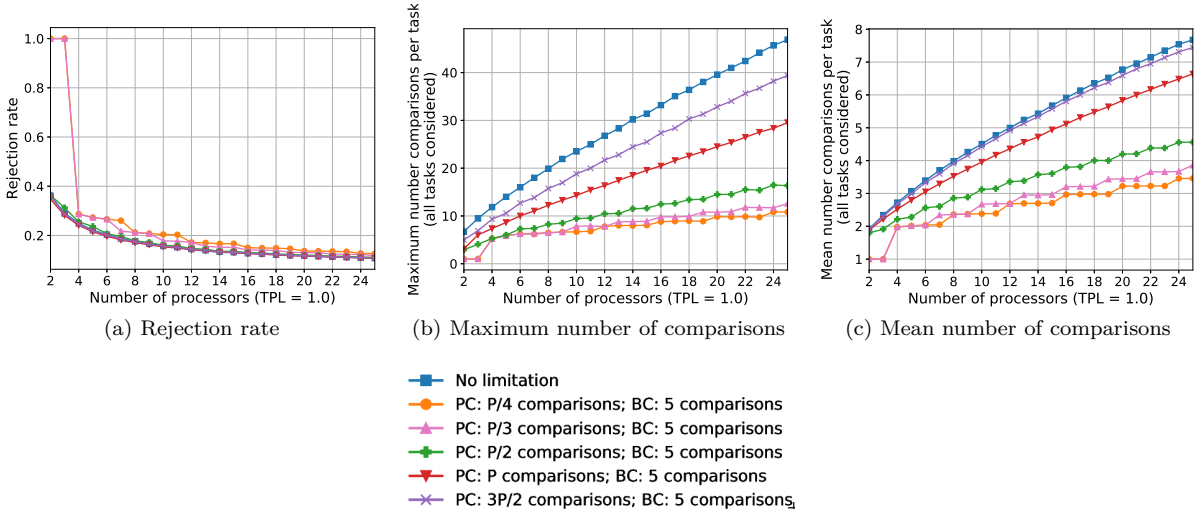
Regarding the number of comparisons, the BSST generally requires more comparisons than the FSST. If we take an example of a 14-processor system using the ES, the mean number of comparisons per task

is increased by 13% and the maximum one by 23% for the PB approach with BC deallocation. For the PB approach with BC deallocation and BC overloading, both numbers of comparisons of the BSST ES are raised by 130% compared to the FSST + ES. This significant difference is due to the higher number of tested slots, as presented in Section 3.1.1.3.

To conclude, the BSST + ES has similar rejection rate as the FSST + ES and the number of comparisons is significantly higher for the BSST. Therefore, the BSST is not a convenient scheduling search technique to reduce the algorithm run-time and it will not be considered in our further work.

3.1.3.9 Limitation on the Number of Comparisons

In this section, we focus on the limitation on the number of comparisons as described in Section 3.1.1.5.



(a) Rejection rate (b) Maximum number of comparisons (c) Mean number of comparisons

Figure 3.22 – Method of limitation on the number of comparisons (PB approach + BC deallocation; FFSS SbS; $TPL = 1.0$)

The presented results are related to the PB approach with BC deallocation and similar results were obtained for the PB approach with BC deallocation and BC overloading. We consider the FFSS SbS and $TPL = 1.0$. Figures 3.22 depict the rejection rate, the maximum and mean numbers of comparisons per task as a function of the number of processors. The value for the backup copies is always set at 5 comparisons and the ones for the primary copies are as follows: $P/4$, $P/3$, $P/2$, P and $3P/2$ comparisons, where P is the number of processors.

Figure 3.22a representing the rejection rate shows that there is almost no difference if more than $P/2$ comparisons for the primary copies are authorised. For systems with 2 and 3 processors and less than $P/2$ comparisons for the primary copies, the rejection rate equals 100% because there are not enough comparisons authorised to schedule a task. Regarding the maximum and mean numbers of comparisons represented respectively in Figures 3.22b and 3.22c, the limitation on the number of comparisons significantly reduces their values as expected.

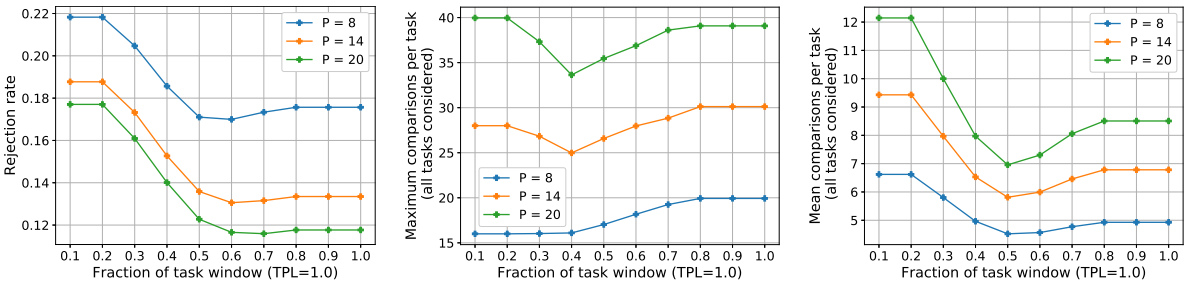
To find a trade-off between the rejection rate and the algorithm run-time, Figure 3.26a plots improvements in the rejection rate and the maximum and mean numbers of comparisons for a 14-processor system. The values of studied metrics are compared to the PB approach without any proposed enhancing method(s) and the higher improvement in %, the better the method.

It can be noticed that, if $P/2$ comparisons for the primary copies is chosen, the rejection rate is deteriorated by 1.50% only compared to the PB approach without this technique and the maximum and mean numbers of comparisons are respectively reduced by 61.9% and 34.21%.

3.1.3.10 Restricted Scheduling Windows

In this section, we analyse the method of restricted scheduling windows. We consider the FFSS SbS, TPL set at 1.0 and the PB approach with BC deallocation. Once again, similar results are obtained for the PB approach with BC deallocation and BC overloading. As an example, we show results respectively conducted for 8, 14 and 20 processors.

Figures 3.23 depict the rejection rate, the maximum and mean numbers of comparisons per task as a function of the fraction of the task window. It can be seen that the represented curves remain constant from $f = 0.1$ to $f = 0.2$ and from $f = 0.8$ to $f = 1.0$. These constant values are due to the minimal considered ratio of the computation times to the task window in our experimental framework, which is $c_i/d_{max,i} = 1/5$ for $2c_i \leqslant d_i \leqslant 5c_i$. Furthermore, we notice that the trend for a given metric is similar regardless of the number of processors. In conformity with the results depicted in Figures 3.15, the more processors, the lower the rejection rate and the more comparisons.



(a) Rejection rate (b) Maximum number of comparisons (c) Mean number of comparisons

Figure 3.23 – Method of restricted scheduling windows (PB approach + BC deallocation; FFSS SbS, $TPL = 1.0$)

Figure 3.23a representing the rejection rate shows that, when the fraction f drops below 0.5, the rejection rate climbs because the scheduling windows become too restrictive. We consequently focus on f between 0.5 and 1 and observe a minimum for $f = 0.6$.

The algorithm run-time is depicted in Figures 3.23b and 3.23c showing the maximum and mean numbers of comparisons per task, respectively. When zeroing in on f between 0.5 and 1, the evolution of the maximum number of comparisons grows with the fraction f of the task window due to more slots to test. When the restricted scheduling windows are fixed at $f = 0.5$, the maximum and mean numbers of comparisons are respectively reduced by 12% and 17% compared to values for $f = 1$. The value of the mean number of comparisons is up to 5.5 times lower when compared with the maximum number of comparisons, which demonstrates that the FFSS SbS does not need to scour all processors all the time to schedule a task.

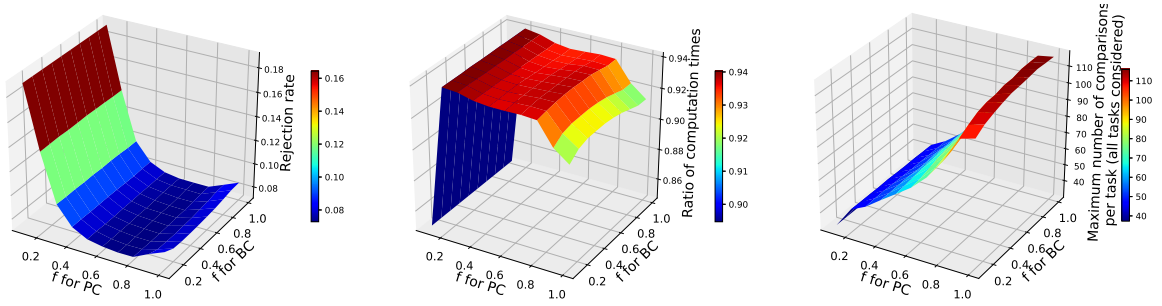
To sum up, the method of restricted scheduling windows diminishes the algorithm run-time, measured by means of the number of comparisons, without notably worsening the system performances, such as the rejection rate. Figure 3.26b represents improvements in the rejection rate and the maximum and mean numbers of comparisons per task for different values of f. It is shown that the reasonable trade-off between the rejection rate and the number of comparisons is obtained for $f = 0.5$ or $f = 0.6$.

PC Scheduling Window versus BC Scheduling Window

So far, we defined the same values of the fraction f for both primary and backup copies. In this section, we consider that the primary and backup copies have different values of f denoted by f_{PC} and f_{BC}, respectively. It means that the restricted scheduling windows for primary and backup copies have different sizes.

The current simulation scenario takes into consideration a 14-processor system and a larger task window, whose size is set at $tw = 11c$ all the time. This modification will ease the observation of metrics because the computation time is inferior to the step of the fraction $f_{PC} = f_{BC} = 0.1$.

Figures 3.24 depict the rejection rate, the ratio of computation times and the maximum number of comparisons as a function of the fractions f_{PC} and f_{BC} of the task window. We consider a 14-processor system using the FFSS PbP, the PB approach with BC deallocation and $TPL = 1.0$. The same results with only minor differences were obtained for PB approach with BC deallocation and BC overloading.



(a) Rejection rate (b) Ratio of computation times (c) Maximum number of comparisons

Figure 3.24 – Method of restricted scheduling windows as a function of the fractions of task window for the primary and backup copies (PB approach + BC deallocation, FFSS SbS; $P = 14$; $TPL = 1.0$)

First of all, it can be seen that all metrics mainly depend on the fraction of task window for the primary copies and they have hardly any dependency on the fraction of task window for the backup ones. Actually, it is easier to place a backup copy than a primary one.

Second, when the fraction of the task window for the primary copies increases, the number of comparisons (the maximum number of comparisons per task depicted in Figure 3.24c) corresponding to the algorithm run-time and the mean time to next fault standing for the system fault tolerance increase, which demonstrates the merit of using the restricted scheduling windows. Although the fraction of task window for the backup copies has little influence in general, its variations in the number of comparisons are noticeable. For example, these variations are up to about 10% for the maximum number of comparisons and about 20% for the mean number of comparisons for the PB approach with BC deallocation.

The rejection rate (Figure 3.24a) and the ratio of computation times (Figure 3.24b) show that the best performances, i.e. the lowest rejection rate and the highest ratio of computation times, are obtained when the fraction of the task window for the primary copies f_{PC} is in range from 0.4 to 0.7. The choice of f_{PC} is therefore again a trade-off among several criteria.

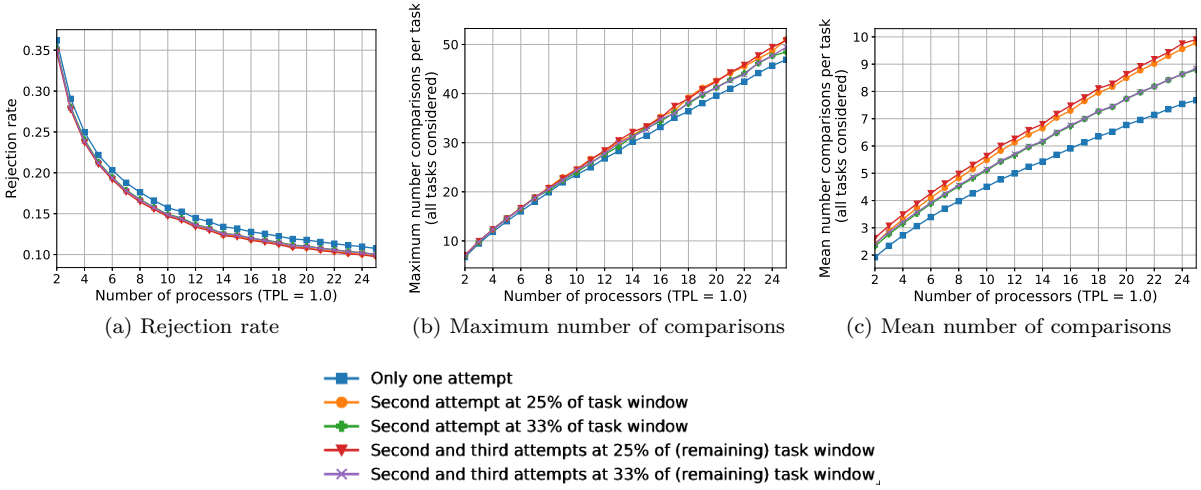
Experiments when $TPL = 0.5$ were also carried out and analysed. The results showed that studied metrics have similar shape to the results when $TPL = 1.0$. Actually, their performances are the same or better because the system workload is lower and the system can accept more tasks.

Thus, the use of restricted scheduling windows for the primary and backup copies is beneficial not only to significantly reduce the algorithm run-time but also to improve the system schedulability. Although the fraction of task window for the primary copies plays a more important role than the one for the backup copies, both are useful. Consequently, the choice of values for f_{PC} and f_{BC} depends on the system application and its constraints. In general, a reasonable trade-off among different parameters is obtained when $f_{PC} = f_{BC} = 0.5$.

3.1.3.11 Several Scheduling Attempts

To evaluate the performances of several scheduling attempts, we make use of the FFSS SbS and set TPL at 1.0. Based on mean values of simulation parameters from Table 3.2, the value of the percentage of slack within the task window ps, as defined in Equation 3.3, is 42.8%, which means that there is a high chance of successful scheduling a task later than at its task arrival. We present results for the PB approach with BC deallocation but it should be noticed that results remain valid for the PB approach with BC deallocation and BC overloading as well. Figures 3.25 depict the rejection rate, the maximum

and mean numbers of comparisons per task as a function of the number of processors when $\omega = 25\%$ and $\omega = 33\%$.

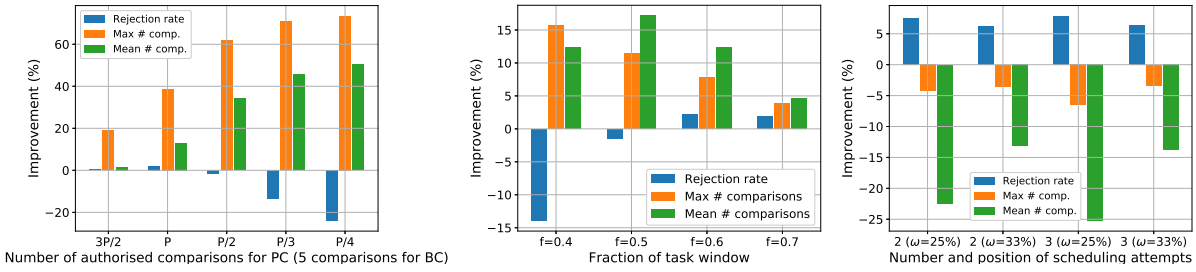


(a) Rejection rate (b) Maximum number of comparisons (c) Mean number of comparisons

Figure 3.25 – Method of several scheduling attempts (PB approach + BC deallocation, FFSS SbS, $TPL = 1.0$)

As it can be seen in Figure 3.25a, two or three scheduling attempts are always beneficial and the decrease in the rejection rate is respectively about 6% or 7%. The maximum and mean numbers of comparisons per task, respectively depicted in Figures 3.25b and 3.25c, are worsened when compared to the algorithm carrying out only one scheduling attempt because every new attempt requires additional comparisons.

Figure 3.26c showing the improvement for a 14-processor system sums up the results. It is worth noticing it is no good trying more than two attempts because we can hardly expect any improvement in the rejection rate and the number of comparisons is higher. The reasonable trade-off between the rejection rate and the number of comparisons is the use of two scheduling attempts at 33% of the task window.



(a) Limitation on the number of comparisons (b) Restricted scheduling windows (c) Several scheduling attempts

Figure 3.26 – Improvements to a 14-processor system compared to the PB approach without proposed enhancing methods (PB with BC deallocation; FFSS SbS; $TPL = 1.0$)

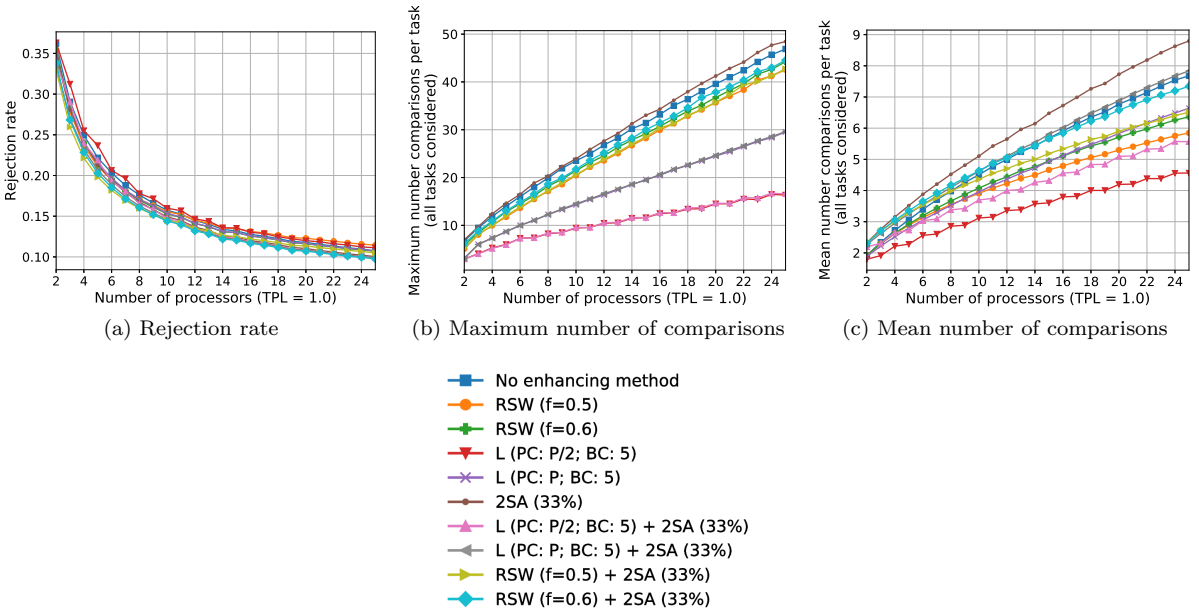
3.1.3.12 Combination of Enhancing Methods

We remind the reader that our aim is to significantly reduce the number of comparisons without worsening the rejection rate. Consequently, we analyse the aforementioned methods (and their combinations) employing the parameters that achieve the best performances from the viewpoint of both the number of comparisons and the rejection rate. The values of these parameters are based on the results

from Sections 3.1.3.9, 3.1.3.10 and 3.1.3.11 and summarised for a 14-processor system in Figures 3.26. The chosen methods (acronyms in square brackets) make use of FFSS SbS, $TPL = 1.0$ and they are as follows:

- Limitation on the number of comparisons (PC: $P/2$ comparisons; BC: 5 comparisons) [L (PC: $P/2$; BC: 5)]
- Limitation on the number of comparisons (PC: P comparisons; BC: 5 comparisons) [L (PC: P; BC: 5)]
- Restricted scheduling windows ($f = 0.5$) [RSW ($f = 0.5$)]
- Restricted scheduling windows ($f = 0.6$) [RSW ($f = 0.6$)]
- Two scheduling attempts at 33% [2SA (33%)]
- Limitation on the number of comparisons (PC: $P/2$ comparisons; BC: 5 comparisons) and two scheduling attempts at 33% [L (PC: $P/2$; BC: 5) + 2SA (33%)]
- Limitation on the number of comparisons (PC: P comparisons; BC: 5 comparisons) and two scheduling attempts at 33% [L (PC: P; BC: 5) + 2SA (33%)]
- Restricted scheduling windows ($f = 0.5$) and two scheduling attempts at 33% [RSW ($f = 0.5$) + 2SA (33%)]
- Restricted scheduling windows ($f = 0.6$) and two scheduling attempts at 33% [RSW ($f = 0.6$) + 2SA (33%)]

These methods are compared to the baseline method, i.e. the PB approach with BC deallocation based on the FFSS SbS without any proposed enhancing techniques. The obtained results are depicted in Figures 3.27 respectively showing the rejection rate, the maximum and mean numbers of comparisons per task as a function of the number of processors when $TPL = 1.0$. Although only the results for the PB approach with BC deallocation are plotted, the PB approach with BC deallocation and BC overloading achieves similar performances.



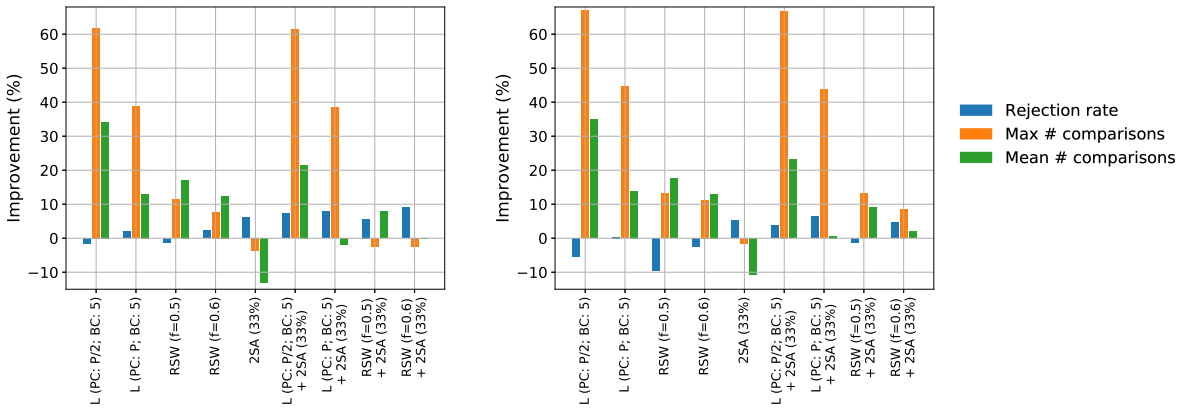
(a) Rejection rate (b) Maximum number of comparisons (c) Mean number of comparisons

Figure 3.27 – Comparison of different methods for the PB approach with BC deallocation as a function of the number of processors (FFSS SbS, $TPL = 1.0$)

Figure 3.27a show that the lowest rejection rate is attained by two scheduling attempts ($\omega = 33\%$) with the limitation on the number of comparisons (PC: P comparisons; BC: 5 comparisons) or with the restricted scheduling windows ($f = 0.6$).

Figure 3.27b illustrates the significant reduction in the maximum number of comparisons per task when the method of limitation on the number of comparisons is put into practice. The mean number of comparisons, represented in Figure 3.27c, is diminished for all methods except when the method of two scheduling attempts is separately put into practice.

To facilitate a comparison among studied techniques, improvements (compared to the PB approach without described techniques) in the rejection rate and in the maximum and mean numbers of comparisons per task are depicted in Figures 3.28. These figures present the PB approach with BC deallocation and with or without BC overloading for the 14-processor system.



(a) PB approach + BC deallocation

(b) PB approach + BC deallocation + BC overloading

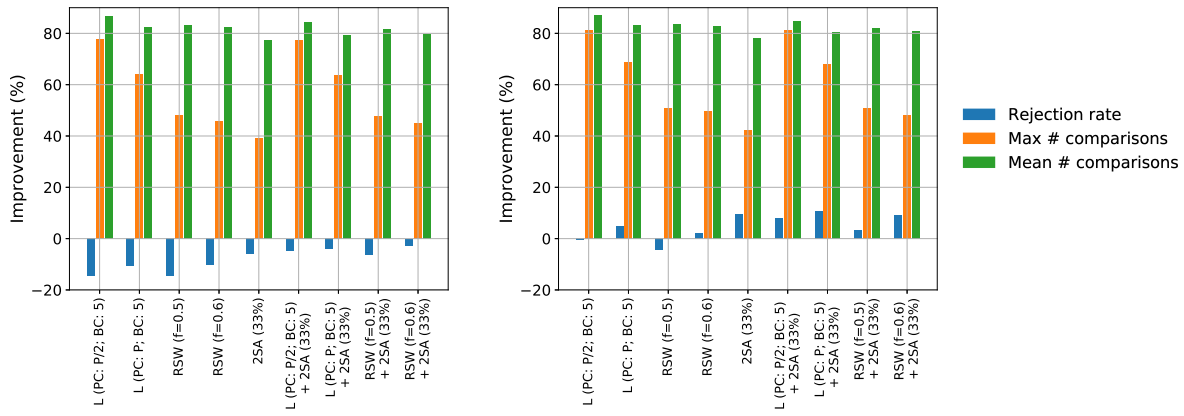
Figure 3.28 – Improvements to a 14-processor system compared to the PB approach without proposed enhancing methods (FFSS SbS; $TPL = 1.0$)

When focusing on the PB approach with BC deallocation (having similar results as the PB approach with BC deallocation and BC overloading), all methods (except when the technique of two scheduling attempts is put into practice separately or in conjunction with the restricted scheduling windows) reduce the number of comparisons and all methods (except the restricted scheduling windows ($f = 0.5$) and the limitation on the number of comparisons (PC: $P/2$ comparisons; BC: 5 comparisons)) decrease the rejection rate. Regardless of the use of the BC overloading, the best methods to reduce both the rejection rate and the number of comparisons are as follows: (i) the limitation on the number of comparisons (PC: $P/2$ comparisons; BC: 5 comparisons) and two scheduling attempts at 33%, and (ii) the limitation on the number of comparisons (PC: P comparisons; BC: 5 comparisons). The number of comparisons of the former technique is reduced by 23% (mean value) and 67% (maximum value) and its rejection rate is decreased by 4% compared to the primary/backup approach without any enhancing method(s).

Moreover, we also compare these two methods to the approach based on the exhaustive search (ES) because it is the method which provides the lowest rejection rate (see Section 3.1.3.5). The results are plotted in Figures 3.29 depicting the PB approach with BC deallocation and with or without BC overloading for the 14-processor system. Whereas the rejection rate is respectively deteriorated by 4.6% and 4.0%, the improvement in the maximum (77% and 64%, respectively) and mean (84% and 79%, respectively) numbers of comparisons are significant and interesting for embedded systems.

3.1.3.13 Simulations with Fault Injection

This section evaluates the fault tolerance performances of the algorithm based on the FFSS SbS for the PB approach with BC deallocation. We consider that the methods of limitation on the number of comparisons (PC: $P/2$ comparisons; BC: 5 comparisons) and two scheduling attempts at 33%, which is the best combination of the enhancing methods studied previously, are put into practice and $TPL = 1.0$.



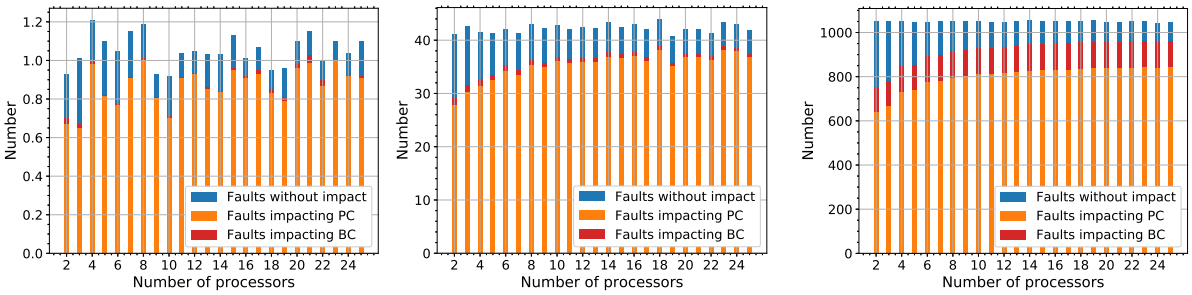
(a) PB approach + BC deallocation

(b) PB approach + BC deallocation + BC overloading

Figure 3.29 – Improvements to a 14-processor system using FFSS SbS compared to the PB approach using ES without proposed enhancing methods ($TPL = 1.0$)

It should be noticed that the conclusions made for this case are also valid for other techniques with and without BC deallocation and/or BC overloading. The only difference is related to the rejection rate when the BC deallocation is not put into practice. Actually, when the BC deallocation is not used, the rejection rate remains the same regardless of the value of fault rates because all tasks copies are scheduled and no backup copy is deallocated. Consequently, such a system has the same performances from the point of view of the system schedulability.

Figures 3.30 depict the total number of faults against the number of processors, while the total number is the sum of the faults without impact, faults impacting simple tasks and faults impacting double tasks. The fault rates injected per processor and represented in the figures respectively equal $1 \cdot 10^{-5}$ fault/ms (corresponding to the worst estimated fault rate in a harsh environment [118]), $4 \cdot 10^{-4}$ fault/ms (corresponding to the limit of the assumption of only one fault in the system at the same time for a 25-processor system) and $1 \cdot 10^{-2}$ fault/ms.



(a) $1 \cdot 10^{-5}$ fault/ms

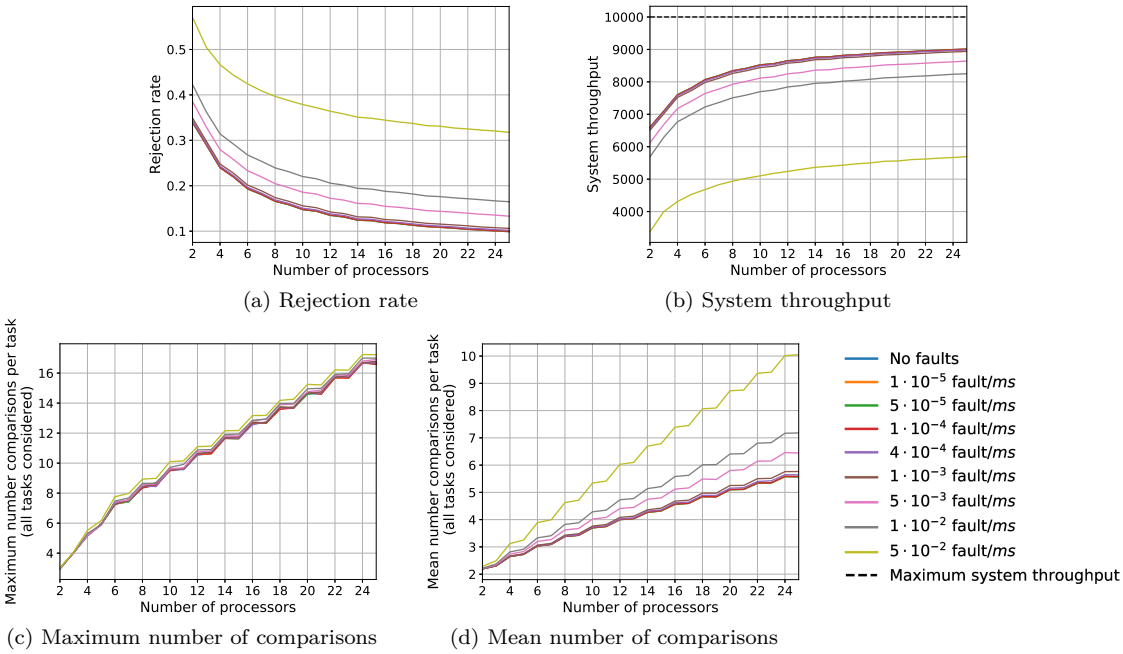
(b) $4 \cdot 10^{-4}$ fault/ms

(c) $1 \cdot 10^{-2}$ fault/ms

Figure 3.30 – Total number of faults (injected with a given fault rate per processor) against the number of processors (PB approach + BC deallocation (FFSS SbS) with limitation on the number of comparisons (PC: $P/2$ comparisons; BC: 5 comparisons) and two scheduling attempts at 33%)

The number of impacted tasks is directly proportional to the processor load, represented in Figure 3.15b. When the rate of injected faults per processor increases, there are more impacted task copies as well. Furthermore, while the number of impacted backup copies is negligible when compared to the one of primary copies, there are more backup copies impacted when the fault injection rates are higher.

It can be seen that the assumption of only one fault in the system at the same time ($4 \cdot 10^{-4}$ fault/ms per processor for a 25-processor system) seems to be a reasonable approximation because the average from 100 simulations shows that backup copies are impacted in 1.3%.



(a) Rejection rate

(b) System throughput

(c) Maximum number of comparisons

(d) Mean number of comparisons

Figure 3.31 – System metrics at different fault injection rates (PB approach + BC deallocation (FFSS SbS) with limitation on the number of comparisons (PC: $P/2$ comparisons; BC: 5 comparisons) and two scheduling attempts at 33%)

Figures 3.31 respectively depict the rejection rate, the system throughput and the maximum and mean numbers of comparisons per task as a function of the number of processors at different fault rates. We remind the reader that the rejection rate characterises the schedulability as described in Section 3.1.2.2, which means that both primary and backup copies are successfully scheduled. Nevertheless, it may happen that a backup copy is impacted by fault too. In this case, such a task does not contribute to the system throughput because it was not correctly executed.

As expected, the higher the fault rate, the higher the rejection rate, the lower the system throughput and the higher the mean number of comparisons. Actually, the more faults occur, the more backup copies need to be executed, which increases the system load and the number of comparisons and reduces the chance of successfully scheduling a task. The maximum number of comparisons remains almost unchanged when compared to the fault-free simulations because the size of the task window remains the same.

We conclude that the algorithm performances do not significantly change up to $1 \cdot 10^{-3}$ fault/ms. This fault rate is higher than the estimated processor fault rate in standard conditions ($2.3 \cdot 10^{-9}$ fault/ms [47]) and even higher than the worst estimated fault rate in a harsh environment ($1 \cdot 10^{-5}$ fault/ms [118]).

3.2 Dependent Tasks

Since there are not only independent tasks but also dependent ones in the real world, we will evaluate how previously studied techniques perform when scheduling dependent tasks.

3.2.1 Assumptions and Scheduling Model

All assumptions and models presented in Section 3.1.1 remain valid.

We call the *application* a set of dependent tasks that can be modelled by a directed acyclic graph (DAG). An example is depicted in Figure 3.32.

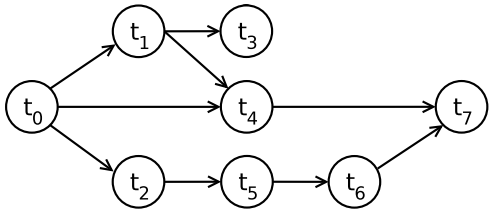


Figure 3.32 – Example of a general directed acyclic graph (DAG)

A DAG is characterised by the nodes and directed edges between the nodes that represent tasks and their dependencies, respectively. The DAG attributes are as follows: arrival time a_{DAG}, deadline d_{DAG}, attributes for each node and attributes for each edges. The node attributes correspond to the task characteristics defined for independent tasks and which are as reads: arrival time a_i, computation time c_i and deadline d_i. Regarding the attributes related to edges, there are characterised by the origin and destination nodes. In this work, we do not take into account communication times.

According to Graham's notation [66] described in Section 1.1, the studied problem is defined as follows:

$$P; m \mid n = k;\ prec;\ online\ r_j;\ d_j = d;\ p_j = p \mid \text{(check the feasibility of schedule)}$$

which means that k dependent jobs/tasks (characterised by release time r_j, processing time p_j and deadline d_j) arrive online on a system consisting of m parallel identical machines and are scheduled to verify the feasibility of a schedule. The algorithm is online global and clairvoyant.

3.2.2 Scheduling Methods

In this manuscript, we do not introduce scheduling methods solely meant for dependent tasks but we make use of the ones already used for independent tasks. This section overviews the methods we implemented for dependent tasks.

We concluded that both the BC deallocation and the BC overloading improve the schedulability when dealing with independent tasks. Consequently, when scheduling dependent tasks, we only analyse the method using the PB approach with BC deallocation and BC overloading. Regarding the scheduling search techniques and processor allocation policies employed to find a slot big enough to place a task copy, we evaluate the following strategies:

— Free Slot Search Technique + First Found Solution First: Processor by Processor (FSST + FFSS PbP)
— Free Slot Search Technique + First Found Solution First: Slot by Slot (FSST + FFSS SbS)
— Boundary Schedule Search Technique + ES: BC scheduled ASAP (BSST + ES BC ASAP)
— Boundary Schedule Search Technique + ES: BC scheduled with maximum overloading (BSST + ES BC maxOverload)

The description of processor allocation policies is given in Section 3.1.1.2 and the principle of scheduling search techniques is presented in Section 3.1.1.3. There is no difference in principles of these methods when scheduling independent or dependent tasks except one. The only modification is that the backup copies of dependent tasks are placed as soon as possible in order not to delay computations in the case a fault occurs during an execution of a primary copy. Nevertheless, the studied approach remains passive, i.e. the primary and backup copies of the same task cannot be executed at the same time. Table 3.3 sums up the position of task copies for studied methods. A search to find a slot large enough for a task copy

(both PC and BC) starts on the processor following the processor on which the last scheduled copy was placed.

Table 3.3 – Task copy position

Method	Primary Copy	Backup Copy
FSST + FFSS PbP	ASAP	ASAP
FSST + FFSS SbS	ASAP	ASAP
BSST + ES BC ASAP	ASAP	ASAP
BSST + ES BC maxOverload	ASAP	Maximise the BC overloading

In order to avoid additional constraints due to strong and weak primary copies, defined in Section 2.6, we consider that, for a task t_j dependent on task t_i, PC_j can be scheduled after the end of both PC_i and BC_i.

3.2.3 Methods to Deal with DAGs

To model tasks dependencies, we make use of directed acyclic graphs (DAGs) generated by DAGGEN. This task graph generator was presented in Section 2.6.2 and more details are available in Appendix B. Every application has its own DAG consisting of several tasks. The main steps of DAG creation are encapsulated in Algorithm 9.

Algorithm 9 Generation of directed acyclic graphs

Input: DAG parameters (number of tasks, fat, density, regularity and jump)
Output: Set of DAGs
1: **for** each application **do**
2: Generate computation times independent of TPL and P
3: Generate a DAG using the parameters set by user (number of tasks, fat, density, regularity and jump)
4: Assign computation times to tasks in the DAG

Once task dependencies are modelled, mapping and scheduling of applications can be conducted. The main steps are summarised in Algorithm 10. First (Lines 1-3), generated DAGs are read, assigned their arrival times and deadlines, and their paths are ordered in decreasing order of their sum of computation times. Then, for each application, start time s_i and deadline d_i are assigned for all tasks according to rules presented in Algorithm 12 and a scheduling search is carried out (Lines 6-8). If all primary and backup copies of all tasks are successfully scheduled, the application is committed, it is rejected otherwise (Lines 9-12). In order to save the algorithm run-time, once a task copy of an application cannot be scheduled, the search finishes and the application is rejected.

Before explaining how start times and deadlines are assigned, we present the algorithm of function `forward_method`. As Algorithm 11 shows, this function determines the deadline of a given task knowing its start time, its multiple defining the size of the task window, and its computation time. This method is called *forward* because the computation is based on the time data preceding the deadline, which is therefore ahead in time compared to the input data.

To assign start times and deadlines to tasks in a DAG, we were inspired by the method, published in [42] and presented in Section 2.6.

The *source task* is the task without any predecessor and the *sink task* is the task without any successor. We call the *known task* the one having already been assigned its start time s_i and deadline d_i. We denote the *segment* that accounts for a part of the path which does not have known tasks. Algorithm 12 encapsulates four cases when computing start times and deadlines. After execution of the algorithm, the task start times and deadlines are uniformly distributed (weighted by computation times) within the available window. All task deadlines are hard. Subsequently, if they are not met, a task cannot be scheduled and the application where a task belongs to is rejected.

Algorithm 10 Main steps to schedule dependent tasks

Input: Set of DAGs
Output: Mapping and scheduling MS of scheduled DAGs
1: Read generated DAGs
2: Compute the length of all paths and sort them in decreasing order
3: Generate DAG arrival time (dependent on TPL and P) and deadline
4: **for** each application **do**
5: Assign start time s_i and deadline d_i for all tasks according to rules presented in Algorithm 12
6: **for** each task **do**
7: Search for PC slot
8: Search for BC slot
9: **if** PC and BC of each task exist **then**
10: Commit the application
11: **else**
12: Reject the application

Algorithm 11 Forward method to determine a deadline

Input: Start time s, Multiple[6] α, Computation time c
Output: Deadline d
1: $d = s + \alpha \cdot c$

In order to improve the schedulability, once PC and BC slots of a task are determined, the start times of their direct children are set. This makes the task windows larger and increases the probability to find a slot large enough to place a task copy.

To illustrate this stage, we consider an application represented by a DAG depicted in Figure 3.33. Each task was given a computation time as noted in the second column in Table 3.4. All task start times and deadlines were computed using Algorithm 12. The results are gathered in Table 3.4.

6. A multiple α is an integer at least equal to 2 in order to be able to schedule both primary and backup copies within the task window without their overlap. The value of α is the same as for the whole DAG.

Algorithm 12 Determination of start times and deadlines of tasks in DAG in our experimental framework

Input: Set of DAGs without assignation of start times and deadlines to tasks
Output: Set of DAGs with assignation of start times and deadlines to tasks

for all paths **do**
  switch type of path **do**
    case A: If no task on the current path has been assigned s_i and d_i, except $s_i(= a_{DAG})$ of the source task and $d_i(= d_{DAG})$ of the sink task of the critical path
      Determine s_i and d_i for all tasks on the current path: $d_i = \texttt{forward\_method}\,(s_i, \alpha, c_i)$

    case B: If the source task of the current path does not have s_i
      if known task t_k exists **then**
        Backward from the known task t_k to determine s_i of the source task:
          $s_i = s_k - \alpha \cdot \sum_{\text{Tasks from } t_k \text{ to } t_i}$ computation time
        For all remaining tasks on the current path: $d_i = \texttt{forward\_method}\,(s_i, \alpha, c_i)$
      else
        Try after scheduling all paths

    case C: If the sink task of the current path does not have d_i
      if known task t_k exists **then**
        Progressively (forward) determine d_i from the known task to the sink task:
          $d_i = \texttt{forward\_method}\,(s_i, \alpha, c_i)$
      else
        Try after scheduling all paths

    case D: Else, i.e. the source and sink tasks of the current path have their s_i and d_i
      Determine s_i and d_i for all remaining tasks:
        $d_i = \texttt{forward\_method}\,(s_i, \beta, c_i)$ where $\beta = \dfrac{d_{\text{end segment}} - s_{\text{start segment}}}{\sum_{\text{All tasks between two known tasks}} \text{computation time}}$

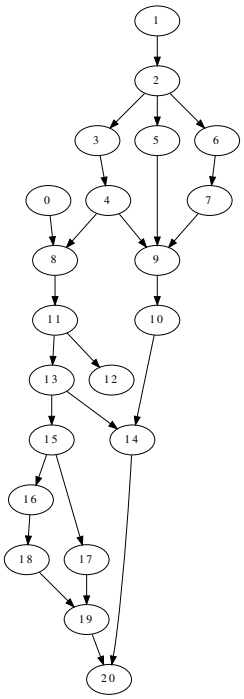


Figure 3.33 – Example of a DAG

Table 3.4 – Example of tasks (belonging to the DAG depicted in Figure 3.33) with their computation times and assigned start times and deadlines

Task t_i	**Computation time** c_i	**Start time** s_i	**Deadline** d_i
0	65	$d_0 - \alpha \cdot c_0$	$s_{20} - \alpha \cdot (c_8 + c_{11} + c_{13} + c_{15} + c_{17} + c_{19})$
1	15	a_{DAG}	`forward_method` (s_1, α, c_1)
2	1 5	$end(BC_1)$	`forward_method` (s_2, α, c_2)
3	15	$end(BC_2)$	`forward_method` (s_3, α, c_3)
4	15	$end(BC_3)$	`forward_method` (s_4, α, c_4)
5	7	$end(BC_2)$	s_9
6	3	$end(BC_2)$	`forward_method` $\left(s_6, \frac{s_9 - d_2}{c_6 + c_7}, c_6\right)$
7	2	$end(BC_6)$	s_9
8	5	$end(BC_0)$	`forward_method` (s_8, α, c_8)
9	15	$end(BC_4)$	`forward_method` (s_9, α, c_9)
10	15	$end(BC_9)$	`forward_method` (s_{10}, α, c_{10})
11	40	$end(BC_8)$	`forward_method` (s_{11}, α, c_{11})
12	15	$end(BC_{11})$	`forward_method` (s_{12}, α, c_{12})
13	5	$end(BC_{11})$	`forward_method` (s_{13}, α, c_{13})
14	15	$end(BC_{10})$	`forward_method` (s_{14}, α, c_{14})
15	4	$end(BC_{13})$	`forward_method` (s_{15}, α, c_{15})
16	3	$end(BC_{15})$	`forward_method` $\left(s_{16}, \frac{s_{19} - d_{15}}{c_{16} + c_{18}}, c_{16}\right)$
17	5	$end(BC_{15})$	`forward_method` (s_{17}, α, c_{17})
18	1	$end(BC_{18})$	s_{19}
19	5	$end(BC_{17})$	`forward_method` (s_{19}, α, c_{19})
20	15	$end(BC_{14})$	d_{DAG}

3.2.4 Experimental Framework

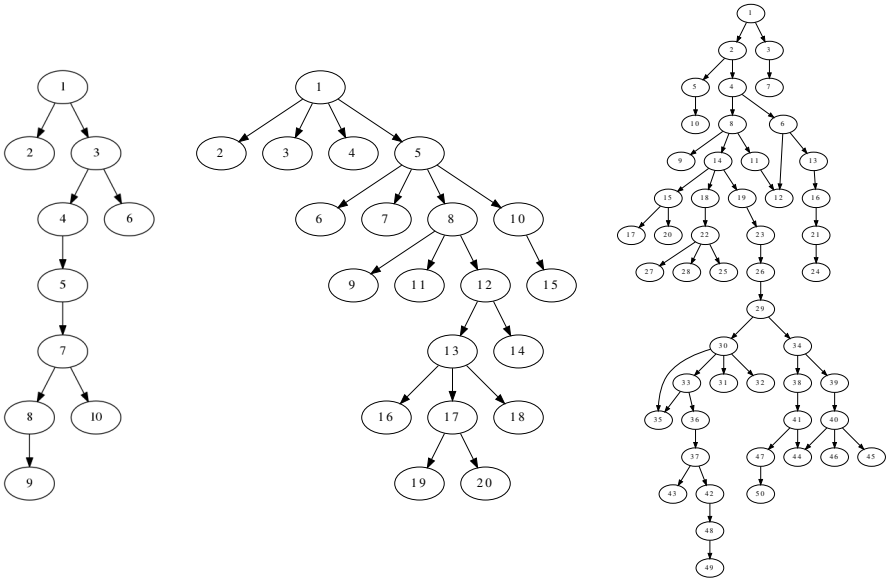
In this section, we describe our simulation scenario and define metrics used to evaluate the algorithms.

3.2.4.1 Simulation Scenario

To generate the directed acyclic graphs (DAGs), we make use of DAGGEN, which is a synthetic task graph generator presented in Section 2.6.2 and Appendix B. The DAG parameters are summarised in Table 3.5. Figures 3.34 depict three examples of DAGs containing respectively 10, 20 and 50 tasks.

Table 3.5 – Parameters to generate DAGs

Parameter	Value
Fat (=width)	0.25
Density	0.5
Regularity	0.1
Jump	3



(a) 10 tasks (b) 20 tasks (c) 50 tasks

Figure 3.34 – Example of generated DAGs

Table 3.6 sums up the simulation parameters used in our experimental framework. For each simulation scenario, 10 simulations of 500 DAGs were treated and the obtained values were averaged. Unless simulation with fault injection are carried out (see Section 3.2.5.7), we consider that no fault occurs during simulations and all backup copies are deallocated when their respective primary copies finish.

The arrival times are generated using the Poisson distribution with parameter λ as follows:

$$\lambda = \frac{\sum^{\text{all tasks in DAG}} c}{TPL \cdot P} \tag{3.5}$$

We remind the reader that, if the *Targeted Processor Load* (TPL) equals 1.0, the arrival times are generated so that every processor is considered to be working all the time at 100%.

Table 3.6 – Simulation parameters

Parameter	Distribution	Value(s)
Number of processors P	-	2 – 25
Number of tasks in one DAG N	-	2; 10; 20; 30; 40; 50; 100
Task computation time c	Uniform	1 – 20 (ms)
Targeted processor load TPL	-	0.50; 1.00
DAG arrival time a_{DAG}	Poisson	$\lambda = \frac{\sum^{\text{all tasks in DAG}} c}{TPL \cdot P}$ (ms)
Size of task window $\sim$ multiple α of the task c	Uniform	2; 5; 7; 10
DAG deadline d_{DAG}	Uniform	$\alpha \cdot$ critical path

To inject faults, we proceed as for the independent task as described in Section 3.1.2.1. We randomly inject faults at the level of the task copies with fault rate for each processor between $1 \cdot 10^{-5}$ and $1 \cdot 10^{-2}$ fault/ms in order to assess algorithm performances not only in real conditions but also in a harsher environment.

3.2.4.2 Metrics

The performances of our algorithms were evaluated based on the following metrics. The *rejection rate* is defined as the ratio of rejected DAGs to all arriving DAGs to the system. The *ratio of computation times* is the proportion of the sum of the computation times of accepted DAGs to the sum of the computation times of all arriving DAGs. The *processor load* characterises the utilisation of processors. The *system throughput* counts the number of correctly executed DAGs. In a fault-free environment, this metric is equal to the number of DAGs minus the number of rejected DAGs.

To assess the algorithm run-time, we make use of the *number of comparisons* standing for the number of tested slots. One comparison is added at each comparison of a slot whether it is large enough to accommodate a task copy (PC or BC) on a given processor. All DAGs are taken into account, no matter whether they are finally accepted or rejected.

3.2.5 Results

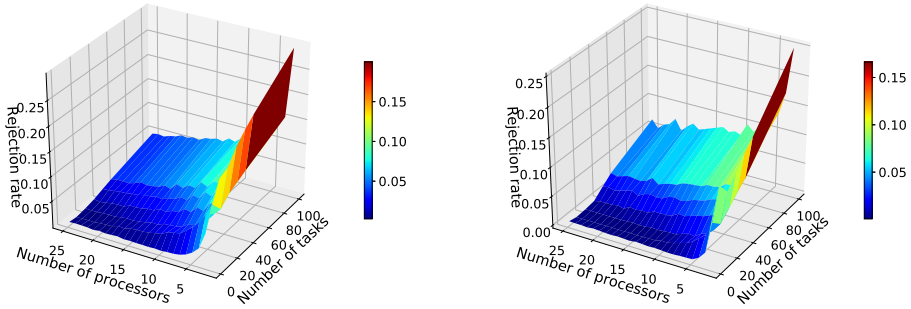
In this section, we evaluate the performances of four techniques (FSST + FFSS PbP, FSST + FFSS SbS, BSST + ES BC ASAP and BSST + ES BC maxOverload) when scheduling dependent tasks. The analyses are based on both 3D and 2D graphs. Finally, we present results with fault injection.

3.2.5.1 3D Graphs: Dependency on the Number of Tasks and the Number of Processors

The results of the rejection rate for the PB approach with BC deallocation and with BC overloading as a function of the number of processors and the number of tasks when $\alpha = 10$ are shown in Figures 3.35 and 3.36 for $TPL = 0.5$ and $TPL = 1.0$, respectively.

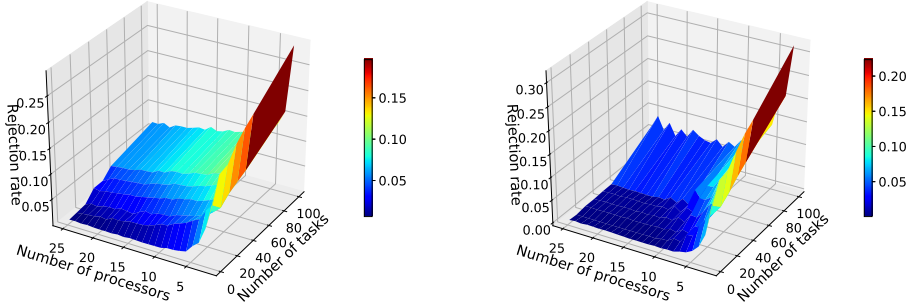
The lower the number of processors and the higher the number of tasks in one DAG, the higher the rejection rate. This phenomenon is to be explained by the facts that (i) the probability to find a slot large enough to place a task is higher when there are more processors, and (ii) the more tasks in one DAG, the more constraints to be satisfied. As expected, the higher the targeted processor load, the higher the rejection rate. We note that while there is almost no difference among studied techniques in the rejection rate for $TPL = 0.5$, the BSST + ES BC maxOverload performs better than the others when $TPL = 1.0$. This difference is due to the search for a slot maximising the BC overloading that improves the schedulability, especially when there are more processors available.

Figures 3.37, 3.38 and 3.39 respectively depict the processor load, the ratio of computation times and the mean number of comparisons per DAG for the PB approach with BC deallocation and with



(a) FSST + FFSS PbP

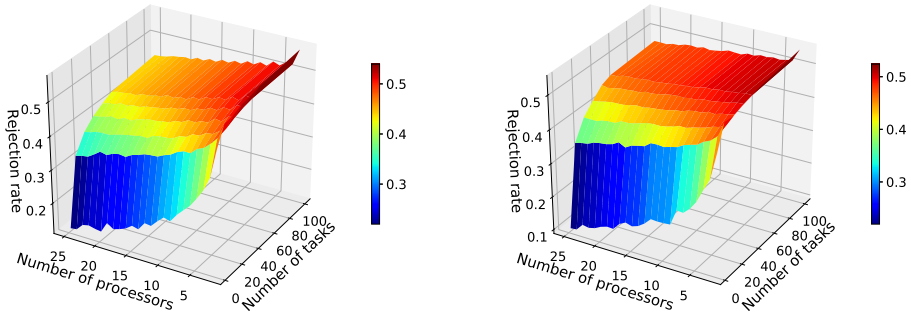
(b) FSST + FFSS SbS



(c) BSST + ES BC ASAP

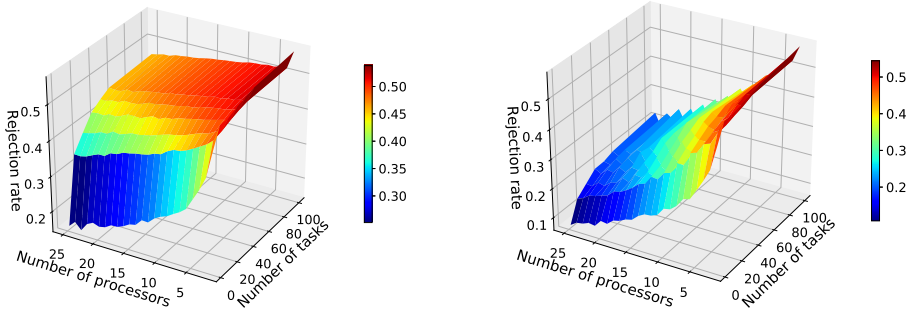
(d) BSST + ES BC maxOverload

Figure 3.35 – Rejection rate as a function of the number of processors and the number of tasks (PB approach + BC deallocation + BC overloading; $TPL = 0.5$; $\alpha = 10$)



(a) FSST + FFSS PbP

(b) FSST + FFSS SbS



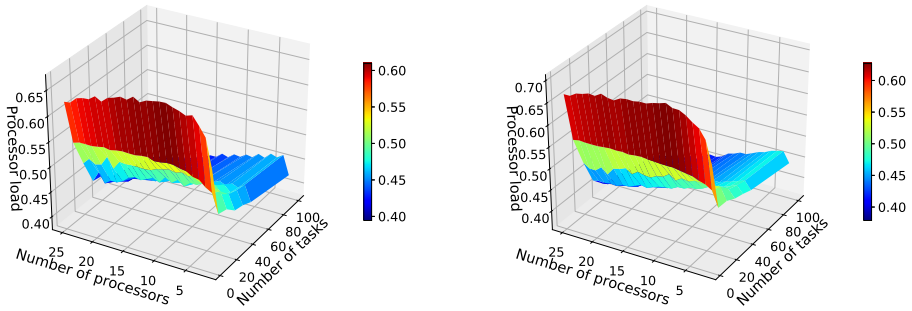
(c) BSST + ES BC ASAP

(d) BSST + ES BC maxOverload

Figure 3.36 – Rejection rate as a function of the number of processors and the number of tasks (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)

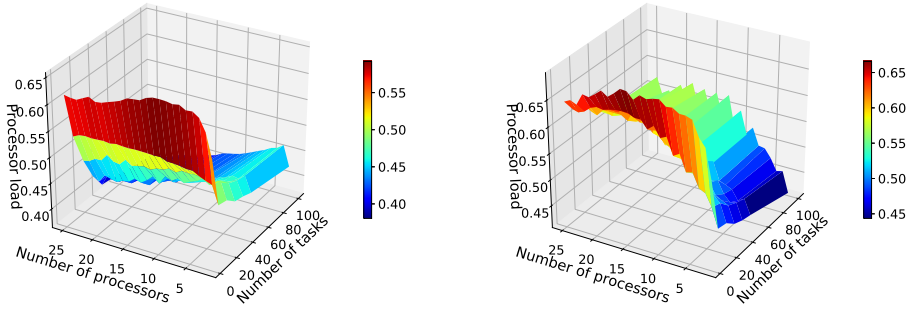
BC overloading as a function of the number of processors and the number of tasks when $\alpha = 10$ and $TPL = 1.0$.

Figures 3.37 show that the lower the number of tasks in one DAG, the higher the processor load due to lower rejection rate. Although DAGs were generated such that $TPL = 1.0$, the real processor load ranges from 40% to 65%. The real processor load is low for DAGs containing more tasks because, if it is not possible to schedule a task in a DAG, the whole DAG is rejected and it thereby contributes to the gap between the targeted processor load and the real one. The same conclusion can be made in Figures 3.38 representing the ratio of computation times, where the values vary from 50% to 90%.



(a) FSST + FFSS PbP

(b) FSST + FFSS SbS



(c) BSST + ES BC ASAP

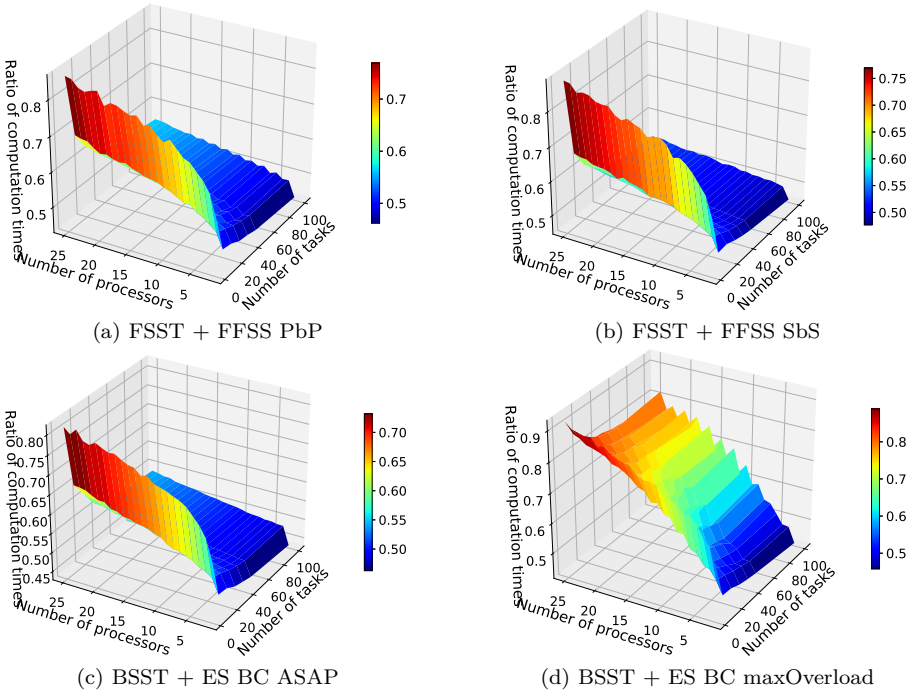
(d) BSST + ES BC maxOverload

Figure 3.37 – Processor load as a function of the number of processors and the number of tasks (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)

Regarding the number of comparisons, there is no qualitative but quantitative difference. In general, the more processors and the more tasks in one DAG, the higher the number of comparisons. Quantitatively, while the searches based on the FSST have at most several tens of thousands of comparisons, the searches using the BSST reach more than two million of comparisons when scheduling DAGs with many tasks on larger systems. This significant difference is caused by the search for a solution. The FSST carries out a search until a solution is found or all processors tested, whereas the BSST always scours all processors to choose the best solution in terms of the position of task copy, as summarised in Table 3.3. Since the number of comparisons accounts for the algorithm run-time, there is a trade-off between this metric and the rejection rate.

3.2.5.2 3D Graphs: Dependency on the Number of Processors and the Size of the Task Window

We evaluate the dependencies of the rejection rate, ratio of computation times and mean number of comparisons per DAG on the number of processors and the size of the task window. The results for the PB approach with BC deallocation and with BC overloading when $\alpha = 10$ and $TPL = 1.0$ are respectively depicted in Figures 3.40, 3.41 and 3.42.



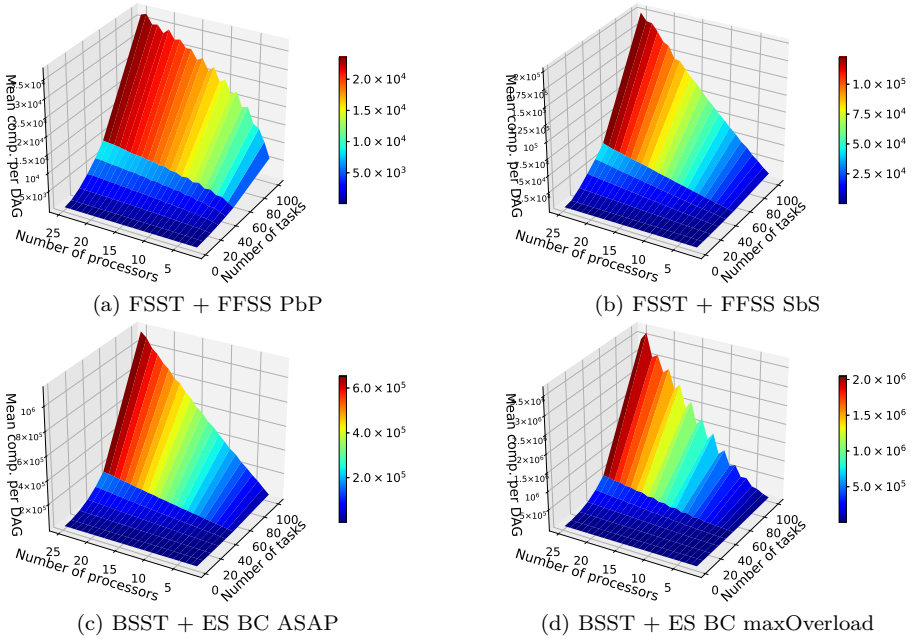
(a) FSST + FFSS PbP

(b) FSST + FFSS SbS

(c) BSST + ES BC ASAP

(d) BSST + ES BC maxOverload

Figure 3.38 – Ratio of computation times as a function of the number of processors and the number of tasks (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)



(a) FSST + FFSS PbP

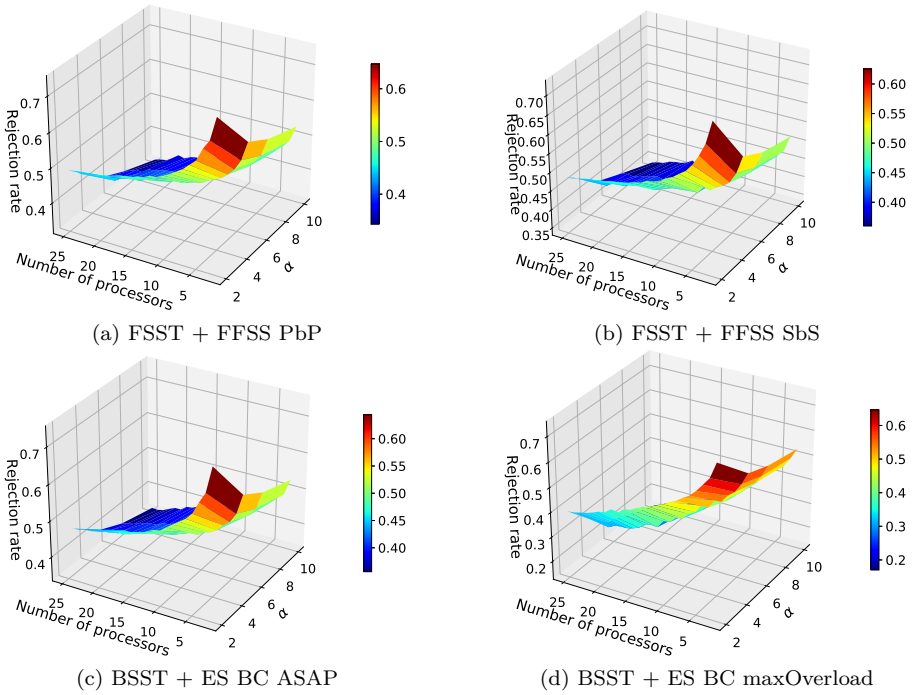
(b) FSST + FFSS SbS

(c) BSST + ES BC ASAP

(d) BSST + ES BC maxOverload

Figure 3.39 – Mean number of comparisons per DAG (all DAGs considered) as a function of the number of processors and the number of tasks (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)

We observe that the lower the number of processors and the smaller the task window, (i) the higher the rejection rate, (ii) the lower the ratio of computation times, and (iii) the lower the number of comparisons. The dependency on the number of processors has already been explained in the preceding section. As regards the dependency on the size of the task window, the larger the task window, the higher probability to find a slot large enough to accommodate a task copy. This yields better system performances (lower rejection rate and higher ratio of computation times) but at the cost of higher algorithm run-time (higher number of comparisons). Again, the BSST + ES BC maxOverload has lower task rejection (when the task window is large and DAGs have more tasks) than other studied techniques.



(a) FSST + FFSS PbP

(b) FSST + FFSS SbS

(c) BSST + ES BC ASAP

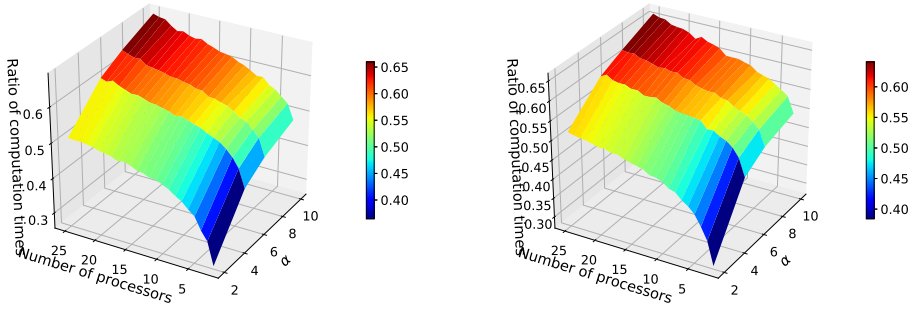
(d) BSST + ES BC maxOverload

Figure 3.40 – Rejection rate as a function of the number of processors and the size of the task window (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)

3.2.5.3 2D Graphs: Dependency on the Number of Processors

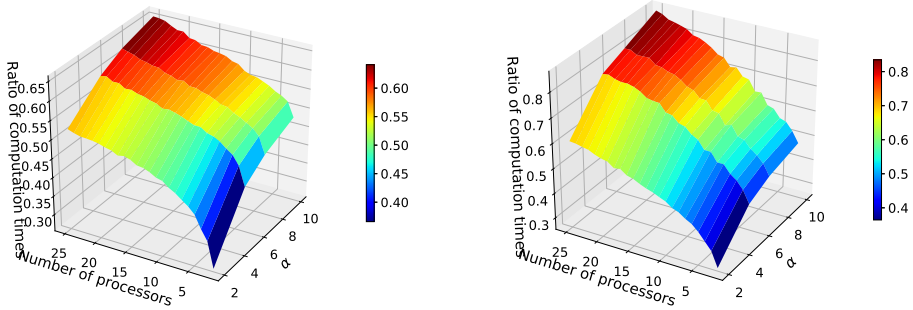
In the preceding sections, we analysed the dependency of the studied metrics in three dimensions. The merit of this visualisation is that it is possible to easier apprehend the evolution of metrics on two parameters at the same time. This is convenient especially for dependent tasks having several parameters, such as the number of tasks in one DAG or the size of the task window. Nonetheless, the 3D representation is not well appropriate to compare different techniques. This is the reason why we analyse also two dimensional graphical representations.

The results of the rejection rate for the PB approach with BC deallocation and with BC overloading as a function of the number of processors when $\alpha = 10$ are represented in Figures 3.43 and 3.44 for $TPL = 0.5$ and $TPL = 1.0$, respectively. The value of the targeted processor load has a significant impact on the rejection rate: the higher its value, the higher the rejection rate. When $TPL = 0.5$, all techniques have almost similar performances although the FSST + FFSS SbS and BSST + ES BC maxOverload perform slightly better for DAGs consisting of only several tasks. For $TPL = 1.0$, when there are less than 5 processors, there is no difference among techniques but starting with 6 processors the gap between the BSST + ES BC maxOverload and other techniques gets larger because the BSST



(a) FSST + FFSS PbP

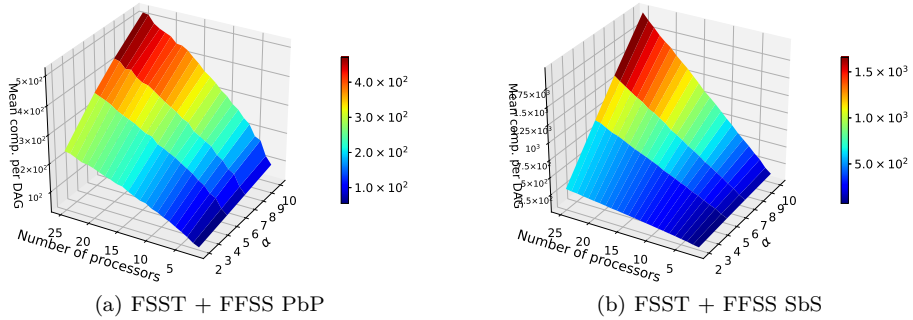
(b) FSST + FFSS SbS



(c) BSST + ES BC ASAP

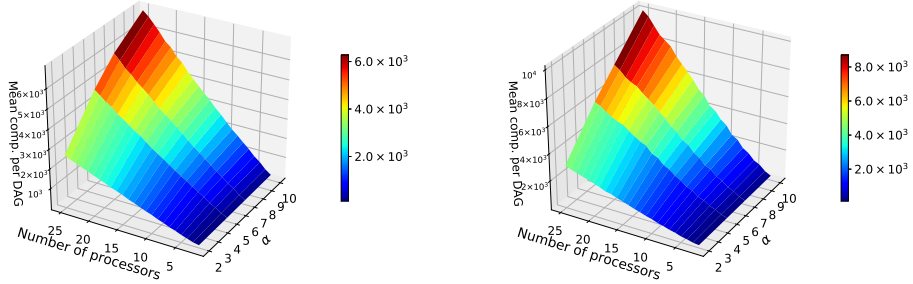
(d) BSST + ES BC maxOverload

Figure 3.41 – Ratio of computation times as a function of the number of processors and the size of the task window (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)



(a) FSST + FFSS PbP

(b) FSST + FFSS SbS

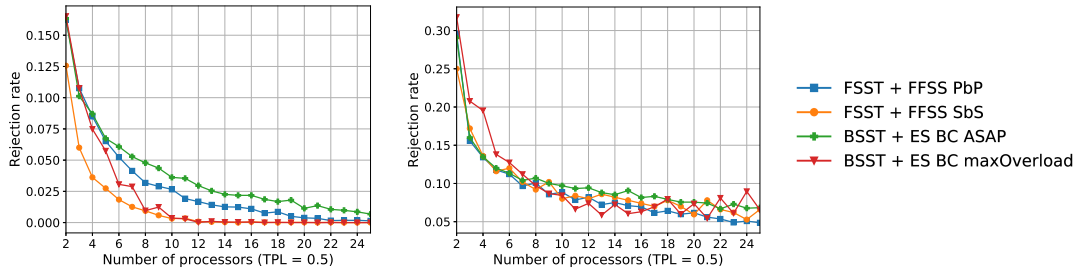


(c) BSST + ES BC ASAP

(d) BSST + ES BC maxOverload

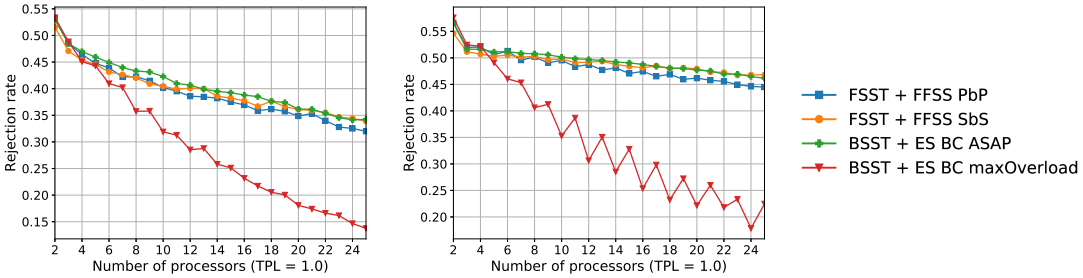
Figure 3.42 – Mean number of comparisons per DAG (all DAGs are considered) as a function of the number of processors and the size of the task window (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)

+ ES BC maxOverload rejects less tasks than other techniques. This gap is also noticeable for the ratio of computation times represented in Figure 3.45.



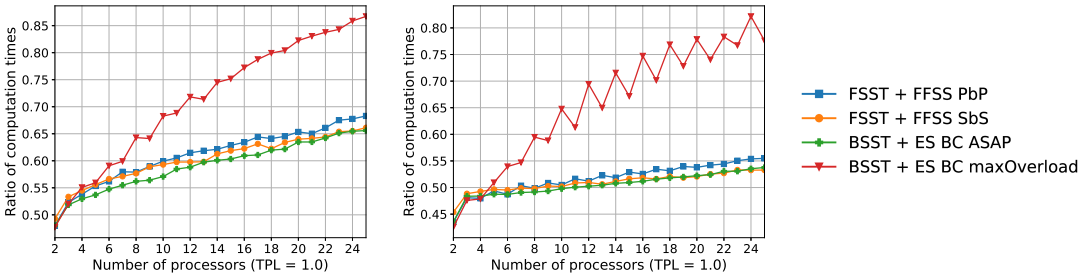
(a) 10 tasks (b) 100 tasks

Figure 3.43 – Rejection rate as a function of the number of processors (PB approach + BC deallocation + BC overloading; $TPL = 0.5$; $\alpha = 10$)



(a) 10 tasks (b) 100 tasks

Figure 3.44 – Rejection rate as a function of the number of processors (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)

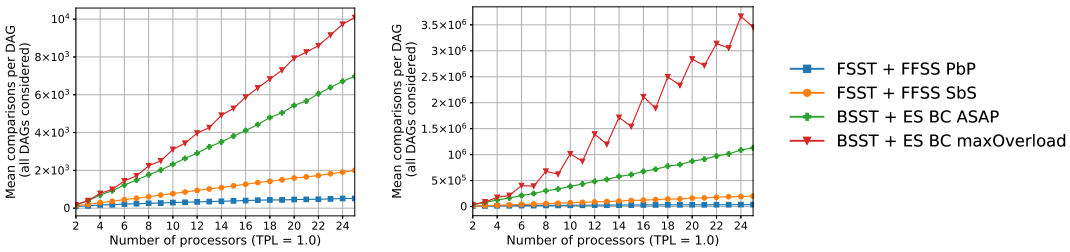


(a) 10 tasks (b) 100 tasks

Figure 3.45 – Ratio of computation times as a function of the number of processors (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)

Regarding the algorithm run-time, the mean number of comparisons per DAG composed of 10 and 100 tasks are depicted in Figures 3.46. The BSST requires significantly more comparisons than the FSST for it always tests all possibilities on all processors. The BSST + ES BC ASAP has less comparisons than the BSST + ES BC maxOverload and the FFSS PbP is quicker than the FFSS SbS. We therefore conclude that, when scheduling dependent tasks, it is better to test all free slots on one processor before trying the next one. The analysis of the maximum number of comparisons per DAG shows that the trend

of curves is similar to the ones plotted in Figures 3.46 but their values are approximately four times higher.



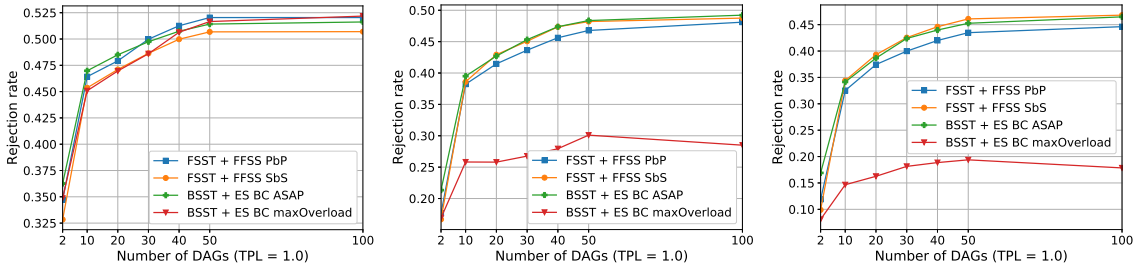
(a) 10 tasks (b) 100 tasks

Figure 3.46 – Mean number of comparisons per DAG (all DAGs are considered) as a function of the number of processors (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)

3.2.5.4 2D Graphs: Dependency on the Number of Tasks

In this section, we evaluate the dependencies of the rejection rate and the mean number of comparisons per DAG on the number of tasks. The results for the PB approach with BC deallocation and with BC overloading when $\alpha = 10$ and $TPL = 1.0$ are respectively depicted in Figures 3.47 and 3.48 for $P \in \{4, 14, 24\}$.

In general, when DAGs contain more tasks, both the rejection rate and the number of comparisons increase and the gap between the BSST and FSST gets larger. The BSST + ES BC maxOverload achieves lower rejection rate than other techniques but at the cost of higher number of comparisons.



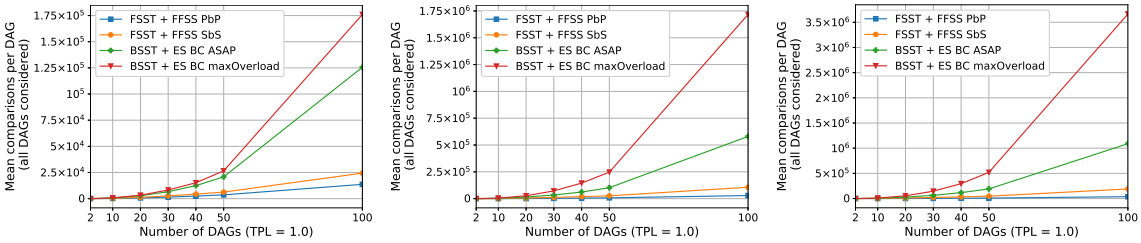
(a) $P = 4$ (b) $P = 14$ (c) $P = 24$

Figure 3.47 – Rejection rate as a function of the number of tasks (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)

3.2.5.5 2D Graphs: Dependency on the Size of the Task Window

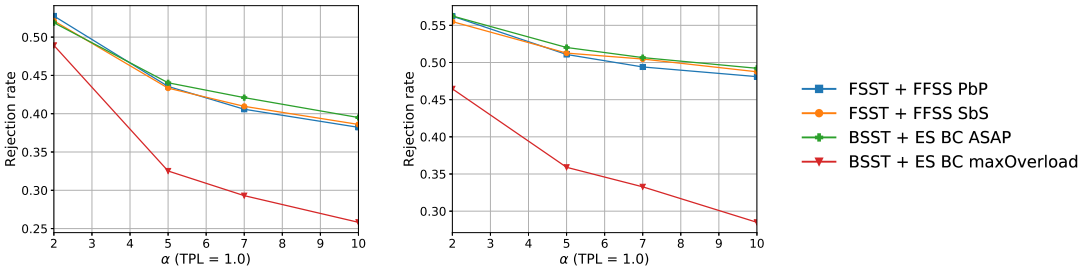
We assess the dependencies of the rejection rate and the mean number of comparisons per DAG on the size of the task window. The results for the PB approach with BC deallocation and with BC overloading when $\alpha = 10$, $P = 14$ and $TPL = 1.0$ are respectively depicted in Figures 3.49 and 3.50 for 10 and 100 tasks in one DAG.

When the size of the task window increases, i.e. when the multiple of the computation times is greater, the lower the rejection rate, the higher the number of comparisons and the larger the gap between the BSST and the FSST due to more possibilities tested.



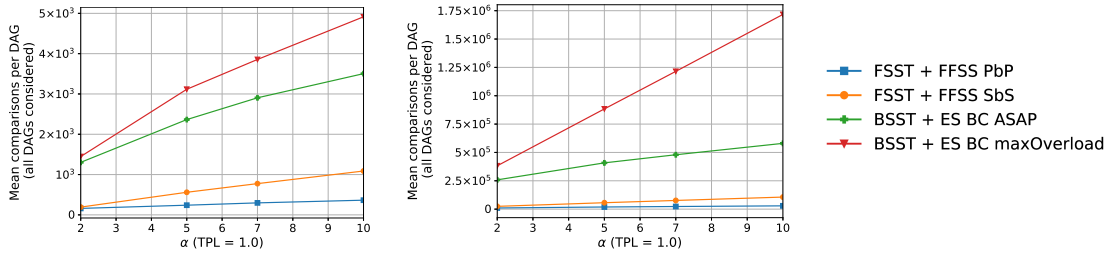
(a) $P = 4$ (b) $P = 14$ (c) $P = 24$

Figure 3.48 – Mean number of comparisons per DAG (all DAGs are considered) as a function of the number of tasks (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $\alpha = 10$)



(a) 10 tasks (b) 100 tasks

Figure 3.49 – Rejection rate as a function of the size of the task window (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $P = 14$; $\alpha = 10$)



(a) 10 tasks (b) 100 tasks

Figure 3.50 – Mean number of comparisons per DAG (all DAGs are considered) as a function of the size of the task window (PB approach + BC deallocation + BC overloading; $TPL = 1.0$; $P = 14$; $\alpha = 10$)

3.2.5.6 Comparison with Already Published Results

After presenting our results, we draw a comparison with results already published in papers. The BSST + ES BC ASAP is close to the online method in [155], which is an update of an offline method presented in [121]. The BSST + ES BC maxOverload is similar to another method published in [155].

The difference between our implementation of the BSST and the one in [155] is that while a primary copy can start before a backup copy of their predecessors (but after their respective primary copies) in [155], in our experimental framework all task copies of predecessors must be finished before a successor task can start its execution. The reason for doing so in our implementation is to reduce the scheduling constraints and consequently avoid longer algorithm time.

Table 3.7 compares two aforementioned methods with our results for a 16-processor system. The main difference is in the dependency on the processor load. In fact, when the targeted processor load increases, the rejection rate in [155] remains almost constant while our rejection rate increases, which seems logical

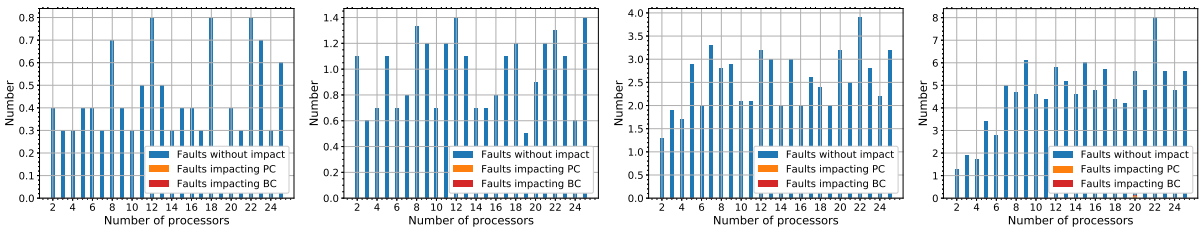
because the higher the targeted processor load, the lower the probability to schedule all tasks. As regards the dependency on the task deadline, both results show that the tighter the deadline, the higher the rejection rate. The obtained values are different, which is probably caused by the different definitions of the task window. While a task window is defined as a multiple $\alpha \in [2;5]$ of the task computation time in our simulations, a task window in [155] is determined as $\eta \cdot \frac{2t_{exe}}{5.5}$, where $\eta \in [0.2;0.3]$, t_{exe} is the execution time of the DAG containing the task and 5.5 is the mean processing speed.

Table 3.7 – Comparison of our results with the ones from [155] for the 16-processor system

	[155]	Our implementation
BSST + ES BC max overloading		
When TPL increases	Rejection rate remains almost the same (mean of 20, 40, 60, 80, 100 tasks: 13% for $TPL \in [0.05;0.8]$)	Rejection rate increases (10 tasks: 0% for $TPL = 0.5$ and 22% for $TPL = 1.0$; 100 tasks: 7% for $TPL = 0.5$ and 25% for $TPL = 1.0$)
When deadline is tighter	Rejection rate increases (mean of 20, 40, 60, 80, 100 tasks: 39% for the smallest studied task window and 3% for the largest studied window)	Rejection rate increases (10 tasks: 47% for the smallest studied task window ($\alpha = 2$) and 23% for the largest studied window ($\alpha = 10$); 100 tasks: 44% for $\alpha = 2$ and 26% for $\alpha = 10$)
BSST +ES BC ASAP		
When TPL increases	Rejection rate remains almost the same (mean of 20, 40, 60, 80, 100 tasks: 15% for $TPL \in [0.05;0.8]$)	Rejection rate increases (10 tasks: 0% for $TPL = 0.5$ and 22% for $TPL = 1.0$; 100 tasks: 7% for $TPL = 0.5$ and 25% for $TPL = 1.0$)
When deadline is tighter	Rejection rate increases (mean of 20, 40, 60, 80, 100 tasks: 40% for the smallest studied task window and 3% for the largest studied window)	Rejection rate increases (10 tasks: 50% for the smallest studied task window ($\alpha = 2$) and 39% for the largest studied window ($\alpha = 10$); 100 tasks: 55% for $\alpha = 2$ and 49% for $\alpha = 10$)

3.2.5.7 Simulations with Fault Injection

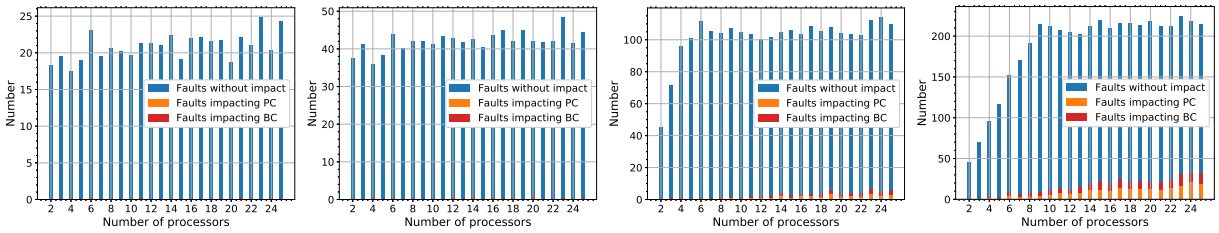
Before presenting the results of different metrics, we carry out a fault analysis. Figures 3.51, 3.52, 3.53 and 3.54 depict the total number of faults against the number of processors, while the total number is the sum of the faults without impact, faults impacting simple tasks and faults impacting double tasks. These figures show such numbers for the PB approach with BC deallocation and with BC overloading when scheduling DAGs consisting of 10, 20, 50 and 100 tasks using the BSST + ES BC maxOverload. The fault rates injected per processor and represented in figures equal $1 \cdot 10^{-5}$ fault/ms (corresponding to the worst estimated fault rate in a harsh environment [118]), $4 \cdot 10^{-4}$ fault/ms (corresponding to the limit of the assumption of only one fault in the system at the same time for a 25-processor system), $1 \cdot 10^{-3}$ fault/ms and $1 \cdot 10^{-2}$ fault/ms.



(a) 10 tasks (b) 20 tasks (c) 50 tasks (d) 100 tasks

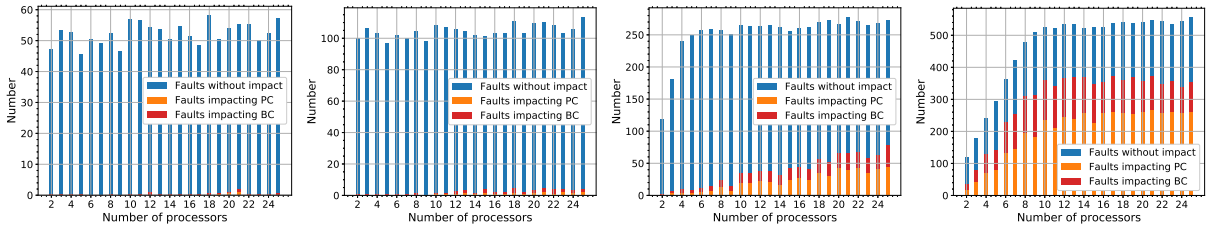
Figure 3.51 – Total number of faults (injected with the fault rate of $1 \cdot 10^{-5}$ fault/ms) against the number of processors (PB approach + BC deallocation + BC overloading; BSST + ES BC maxOverload; $TPL = 1.0$; $\alpha = 10$)

Although results only for the BSST + ES BC maxOverload are shown, other approaches achieve similar values, except when the fault rate is higher than $1 \cdot 10^{-3}$ fault/ms. In such a case, the BSST +



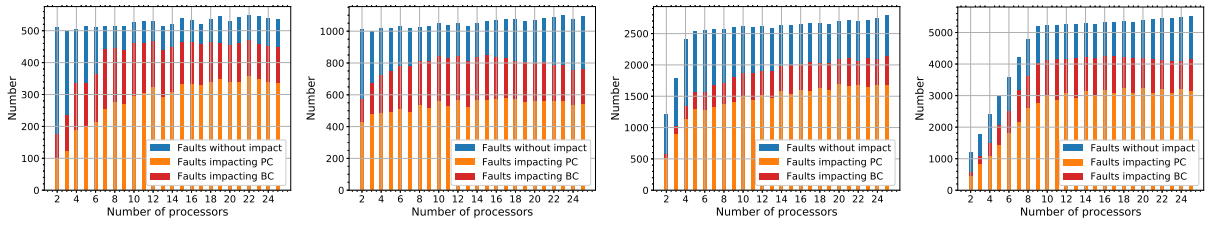
(a) 10 tasks (b) 20 tasks (c) 50 tasks (d) 100 tasks

Figure 3.52 – Total number of faults (injected with the fault rate of $4 \cdot 10^{-4}$ fault/ms) against the number of processors (PB approach + BC deallocation + BC overloading; BSST + ES BC maxOverload; $TPL = 1.0$; $\alpha = 10$)



(a) 10 tasks (b) 20 tasks (c) 50 tasks (d) 100 tasks

Figure 3.53 – Total number of faults (injected with the fault rate of $1 \cdot 10^{-3}$ fault/ms) against the number of processors (PB approach + BC deallocation + BC overloading; BSST + ES BC maxOverload; $TPL = 1.0$; $\alpha = 10$)



(a) 10 tasks (b) 20 tasks (c) 50 tasks (d) 100 tasks

Figure 3.54 – Total number of faults (injected with the fault rate of $1 \cdot 10^{-2}$ fault/ms) against the number of processors (PB approach + BC deallocation + BC overloading; BSST + ES BC maxOverload; $TPL = 1.0$; $\alpha = 10$)

ES BC maxOverload has higher number of impacted copies due to lower rejection rate than the FSST + FFSS SbS and the FSST + FFSS PbP). The BSST + ES BC ASAP is not evaluated because it performs as the FSST but exhibits higher number of comparisons.

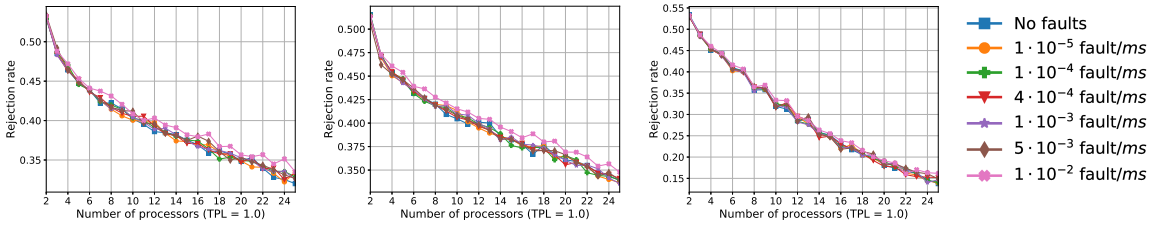
The number of impacted copies increases when a DAG contains more tasks and if faults occur more frequently. Even for the fault rate $1 \cdot 10^{-3}$ fault/ms, which is higher by two orders of magnitude than the worst estimated fault rate, the number of impacted faults is low, except for DAGs containing 100 tasks. This is partially due to the non-negligible rejection rate and therefore lower processor load.

Next, we analyse the rejection rate, the system throughout, the processor load and the mean number of comparisons per DAG for the PB approach with BC deallocation and with BC overloading as a function of the number of processors when $\alpha = 10$ and $TPL = 1.0$. These metrics are respectively depicted in Figures 3.55, 3.57, 3.59 and 3.60 for the FSST + FFSS PbP, the FSST + FFSS SbS and the BSST + ES

BC maxOverload. We focus on the case when each DAG contains 10 tasks but results remain qualitatively valid for DAGs composed of different number of tasks.

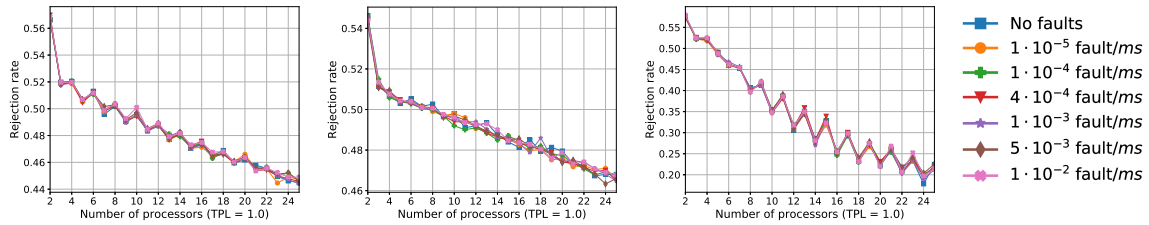
Regarding the rejection rate shown in Figures 3.55 and 3.56, it can be seen that there is almost no difference, except for the FSST with the fault rate $1 \cdot 10^{-2}$ fault/ms and 10 tasks in one DAG, which exhibits slightly higher rejection rate. This is caused by higher number of impacted tasks and their backup copies, which cannot be deallocated. Therefore, the schedulability is only a little impacted by faults and time and space constraints of dependent tasks have predominant effect.

The system throughput shown in Figures 3.57 and 3.58 presents the correct execution of DAGs. The higher the fault rate, the lower this metric starting from the fault rate $5 \cdot 10^{-3}$ fault/ms because the number of impacted backup copies increases. Nevertheless, we conclude the system throughput is not impacted by fault occurrence even in a harsh environment ($1 \cdot 10^{-3}$ fault/ms). It can be seen that the system throughput of the FSST for DAGs with 100 tasks decreases when the number of processors goes from 2 to 10 processors. This phenomenon is explained by the increasing number of impacted primary and backup copies as shown in Figure 3.54d.



(a) FSST + FFSS PbP (b) FSST + FFSS SbS (c) BSST + ES BC maxOverload

Figure 3.55 – Rejection rate at different fault injection rates as a function of the number of processors (PB approach + BC deallocation + BC overloading; 10 tasks in one DAG; $TPL = 1.0$; $\alpha = 10$)

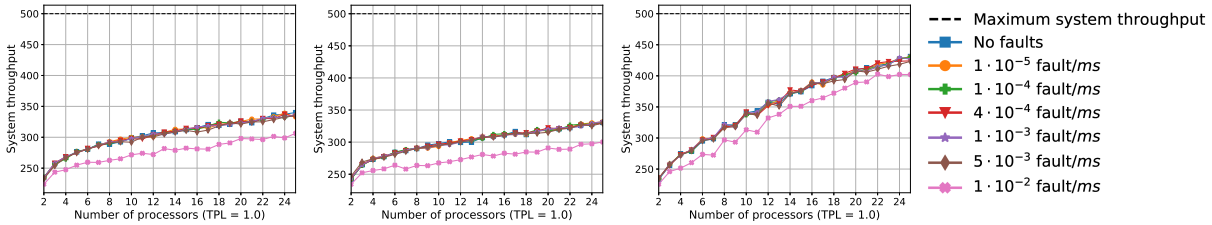


(a) FSST + FFSS PbP (b) FSST + FFSS SbS (c) BSST + ES BC maxOverload

Figure 3.56 – Rejection rate at different fault injection rates as a function of the number of processors (PB approach + BC deallocation + BC overloading; 100 tasks in one DAG; $TPL = 1.0$; $\alpha = 10$)

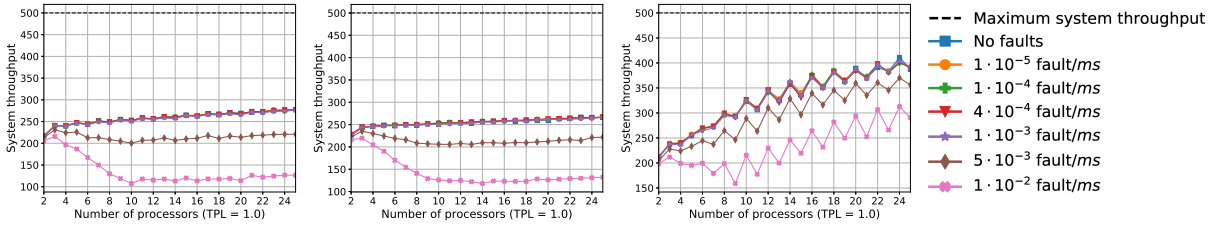
Figures 3.59 depicting the processor load show that the higher the fault rate, the more tasks are impacted and more backup copies need to be executed. While the increase is negligible up to $1 \cdot 10^{-3}$ fault/ms, it starts to be noteworthy for fault rates $5 \cdot 10^{-3}$ fault/ms and $1 \cdot 10^{-2}$ fault/ms. In fact, there are more backup copies that cannot be deallocated and need to be executed because their respective primary copies failed. We note that the processor load, especially for DAGs with 100 tasks, decreases as a function of the number of processors because once a primary or backup copy of any task in a DAG cannot be scheduled, the whole DAG is rejected, which creates a gap between the targeted processor load and the real one.

The further analysis of the system load showed that the number of DAGs that are executed at the same time depends on the system throughput but is independent of the number of tasks in one DAG (for the same system throughput). For example, for simulation parameters ($TPL = 1.0$; $\alpha = 10$; 500 DAGs;



(a) FSST + FFSS PbP (b) FSST + FFSS SbS (c) BSST + ES BC maxOverload

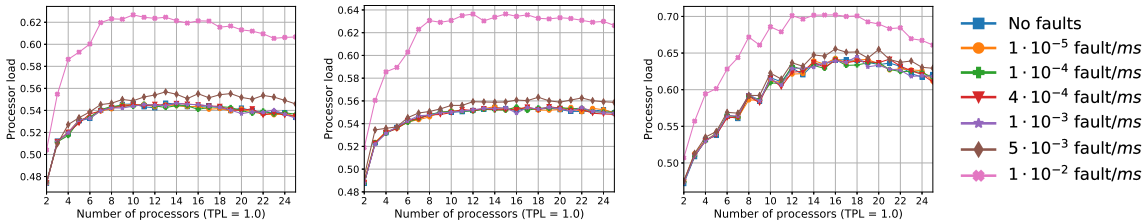
Figure 3.57 – System throughout at different fault injection rates as a function of the number of processors (PB approach + BC deallocation + BC overloading ; 10 tasks in one DAG; $TPL = 1.0$; $\alpha = 10$)



(a) FSST + FFSS PbP (b) FSST + FFSS SbS (c) BSST + ES BC maxOverload

Figure 3.58 – System throughout at different fault injection rates as a function of the number of processors (PB approach + BC deallocation + BC overloading; 100 tasks in one DAG; $TPL = 1.0$; $\alpha = 10$)

FSST + FFSS SbS; no fault injected), they are approximately 2 DAGs for 4-processor system, 6 DAGs for 14-processor system and 9 DAGs for 25-processor system.

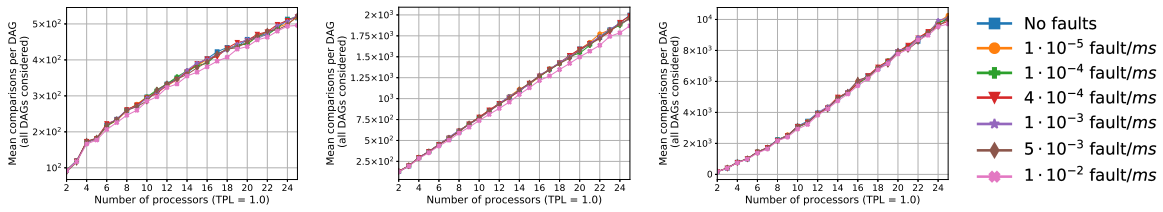


(a) FSST + FFSS PbP (b) FSST + FFSS SbS (c) BSST + ES BC maxOverload

Figure 3.59 – Processor load at different fault injection rates as a function of the number of processors (PB approach + BC deallocation + BC overloading; 10 tasks in one DAG; $TPL = 1.0$; $\alpha = 10$)

Regarding the number of comparisons, all studied fault rates, except $1 \cdot 10^{-2}$ fault/ms for FSST, require similar number of comparisons. Less comparisons for the fault rate of $1 \cdot 10^{-2}$ fault/ms is caused by higher rejection rate. Actually, although before finding out that a primary or backup copy of any task in a DAG cannot be scheduled, all possibilities are tested, the task causing the rejection can be anywhere in the DAG, which lowers the mean number of comparisons per DAG.

The thorough analysis was also carried out for the same simulation parameters when $TPL = 0.5$. The values of the studied metrics were proportional to a system with lower targeted processor load, as it can be seen for the rejection rate of different methods when $TPL = 0.5$ (Figures 3.35) and when $TPL = 1.0$ (Figures 3.36). Nonetheless, the algorithm performances do not change, which means that the schedulability is only a little impacted by faults and that task dependencies have predominant effect.



(a) FSST + FFSS PbP (b) FSST + FFSS SbS (c) BSST + ES BC maxOverload

Figure 3.60 – Mean number of comparisons per one DAG at different fault injection rates as a function of the number of processors (PB approach + BC deallocation + BC overloading; 10 tasks in one DAG; $TPL = 1.0$; $\alpha = 10$)

3.3 Summary

This chapter presents our achievements related to the primary/backup approach for both independent and dependent tasks. They show various aspects of fault tolerant scheduling of aperiodic tasks based on the primary/backup approach.

The next ten paragraphs enumerate the main results of the scheduling of the independent tasks.

First, the results of the PB approach by itself and the one with BC overloading show that the BC overloading facilitates the reduction of the rejection rate (for example 14% for a 14-processor system with $TPL = 1.0$). When the BC deallocation is then put into practice, the improvement is even more noteworthy. For instance for the 14-processor system and $TPL = 1.0$, the gain is about 75% compared to the baseline PB approach and no matter whether the BC overloading is implemented or not. Moreover, it was shown that the BC overloading and the BC deallocation work well together.

Second, the active PB approach was evaluated. It was demonstrated that it allows systems to deal with tasks having tight deadline and it therefore reduces the rejection rate. For example, for the 14-processor system and $TPL = 1.0$, there is a drop of about 17% for both the PB approach with BC deallocation and with or without BC overloading.

Third, three different processor allocation policies were studied: the *exhaustive search* (ES), the *first found solution search - processor by processor* (FFSS PbP) and the *first found solution search - slot by slot* (FFSS SbS). On the one hand, it was found that the ES achieves the lowest rejection rate compared to the other two searches but it has the highest values for the maximum and mean numbers of comparisons per task. On the other hand, the FFSS SbS performs better than the FFSS PbP. While its rejection rate is higher by about 12% compared to the ES (14 processors), the maximum number of comparisons is significantly lower than the one for the FFSS PbP (29% for the 14-processor system) and the ES (about 41% for the 14-processor system). Moreover, the mean number of comparisons is only slightly dependent on the number of processors, which is advantageous to systems with many resources. When comparing the results to the optimal solution, the algorithm based on the FFSS SbS is 2-competitive.

Fourth, two scheduling search techniques were analysed: the *free slot search technique* (FSST) and the *boundary schedule search technique* (BSST). The BSST + ES achieves similar rejection rate as the FSST + ES and the number of comparisons of the BSST is significantly higher than the one of the FSST (more than twice). The BSST is consequently not a convenient scheduling search technique to reduce the algorithm run-time.

Fifth, the overheads of the algorithm based on the primary/backup approach were also analysed. Since this approach reserves slots for its primary and backup copies (even if the BC deallocation is put into practice), the higher the number of processors, the more comparisons to find slots for both copies and consequently the wider the gap in the number of comparisons between fault tolerant and non-fault tolerant systems. It was also shown that the more processors, the narrower the gap in the rejection rate between fault tolerant system using the primary/backup approach and non-fault tolerant one.

The last five paragraphs predominantly cover our achievements for the analysis of the main already existing methods for the PB approach. Although these methods are often put into practice, they have never been analysed and compared. The achievements summarised in the next five paragraphs deal with the proposed enhancements for the PB approach.

Sixth, the method of *limitation on the number of comparisons* was introduced. This very simple method provides interesting results. For example, when the threshold for primary copies is set at $P/2$ comparisons (P denotes the number of processors) and the one for backup copies is fixed at 5 comparisons, the maximum and mean numbers of comparisons per task are respectively cut down by 62% and 34%, whereas the rejection rate is higher by only 1.5% compared to the approach without this technique.

Seventh, another method aiming at reducing the algorithm run-time is technique of the *restricted scheduling windows*. It diminishes the algorithm run-time, measured again by means of the number of comparisons, without worsening the system performances, such as the rejection rate. A reasonable trade-off between the rejection rate and the number of comparisons is obtained for the fraction of task window equal to 0.5 or 0.6.

Eighth, the *several scheduling attempts* method focuses on the reduction in the rejection rate. The results showed that it is useless to carry out more than two scheduling attempts because the rejection rate is not notably better and the number of comparisons per task increases too much. A reasonable trade-off between the rejection rate and the number of comparisons is achieved for two scheduling attempts at 33% of the task window. In such a case, the rejection rate is decreased by 6.2%.

Ninth, we analysed combinations of the aforementioned methods. It was found that almost all the proposed methods diminish the number of comparisons per task and decrease the rejection rate. The best methods to reduce both the rejection rate and the number of comparisons are (i) the limitation on the number of comparisons (PC: $P/2$ comparisons; BC: 5 comparisons) combined with two scheduling attempts at 33%, and (ii) the limitation on the number of comparisons (PC: P comparisons; BC: 5 comparisons). The algorithm run-time of the former technique is reduced by 23% (mean value) and 67% (maximum value) and its rejection rate is decreased by 4% compared to the primary/backup approach without any enhancing method.

Tenth, the results showed 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 [47]) and severe ($1 \cdot 10^{-5}$ fault/ms [118]) conditions. Our algorithm can therefore perform well in a harsh environment.

As regards the dependent tasks, it was shown that when the search for a slot to schedule a task copy is carried out by the BSST + ES, the number of comparisons per application modelled by directed acyclic graph (DAG) is significantly higher than the one based on the FSST + FFSS PbP or FSST + FFSS SbS. Actually, while the BSST + ES scours all processors and tests all free slots, the other two techniques conduct a search until a solution is found or all processors tested. Consequently, the BSST + ES BC maxOverload has better performances than other studied techniques in terms of the rejection rate and system throughout but at the cost of longer algorithm run-time, except for systems with only several processors. Furthermore, the FFSS SbS and FFSS PbP achieve similar performances but the FFSS SbS requires more comparisons.

Last but not least, simulations with fault injection unveil that faults, having fault rates even higher than the worst estimated fault rate in a harsh environment ($1 \cdot 10^{-5}$ fault/ms [118]), have a minimal impact on the scheduling proposed algorithm compared with space and time constraints due to task dependencies.

The achievements of this chapter were published in Proceedings of the 21$^{\text{th}}$ International Workshop on Software and Compilers for Embedded Systems (SCOPES) and of the Conference on Design and Architectures for Signal and Image Processing (DASIP), both held in 2018.

Chapter 4

CubeSats and Space Environment

As it was mentioned in the introduction, the research scope of the PhD thesis is twofold. While the first part is concerned with the primary/backup approach and was treated in the preceding two chapters, the second part deals with fault tolerant scheduling algorithms for small satellites called *CubeSats*. Before presenting our solution to make CubeSats more robust in Chapter 5, this chapter introduces such satellites and the harsh space environment where they operate.

Firstly, we will classify satellites according to their weight and size. Secondly, we will introduce CubeSats. We start with their advent, show their progressive popularity and give some examples of their missions. We will also list main CubeSat systems and tasks executed on board. Thirdly, we present the space environment and how these small satellites are vulnerable to this harsh environment. And fourthly, we sum up methods currently used to provide CubeSats with fault tolerance.

4.1 Satellites

In July 2019, the National Geographic magazine published that there are more than 8 000 man-made objects in outer space and the radars of the U.S. Space Surveillance Network track more than 13 000 objects that are larger than ten centimetres [111]. The website `https://www.n2yo.com/` [1] tracks even 20 721 objects (as of June 1, 2020). 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 [110]:

- 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)
- Femtosatellite (0.001 kg to 0.01 kg)

In order to visualise the difference in weight and size, Figure 4.1 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.

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 TMR described in Section 1.4.

The **yellow ellipse** incorporates for 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.

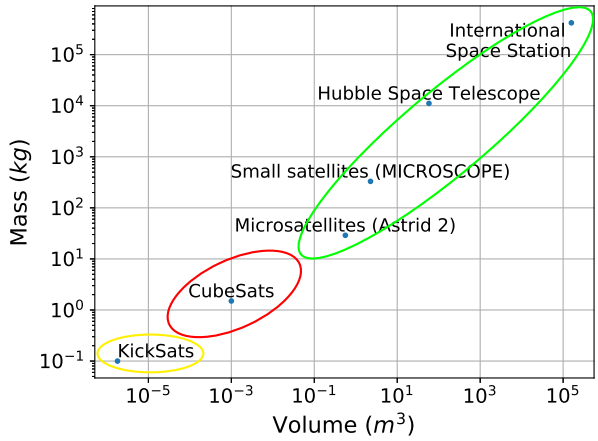


Figure 4.1 – Comparison of satellites

Regarding this trade-off between physical aspects (weight, size and energy) and fault tolerance, CubeSats are in the centre of our interest. Actually, 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 [91] 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.

4.2 CubeSats

CubeSats are small satellites composed of several units [108]. Each unit (1U) is a $10\,cm$ cube weighing up to $1.33kg$. Depending on a particular mission, CubeSats usually consist of 1U, 2U, 3U or 6U. Figure 4.2 depicts Phoenix CubeSat, which is a 3U CubeSat. The CubeSat size does not necessarily scale with the number of tasks and the number of used units mainly depends on the size of payload. Their lifetime is in general 2 or 3 years.

The first CubeSat project began as a collaborative effort in 1999 between Jordi Puig-Suari, a professor from California Polytechnic State University (Cal Poly), and Bob Twiggs, a professor from Stanford University's Space Systems Development Laboratory (SSDL) [108]. The aim of this project consists in providing affordable access to space for universities.

They defined standard parameters in order to reduce costs. In fact, the standardised components can be produced in series and simplify the technical development. Nowadays, there are several companies developing and selling components for CubeSats, e.g. Clyde Space [1], CubeSatShop [2], Pumpkin Space Systems [3] or SkyFox Labs [4].

Last but not least, the standardised dimensions facilitate the deployment of CubeSats into space. When launching a CubeSat, it is attached to a launch vehicle or a rocket and, once the launch vehicle/rocket reaches the desired orbit, the CubeSat is released.

1. `www.clyde.space`
2. `www.cubesatshop.com`
3. `www.pumpkinspace.com`
4. `www.skyfoxlabs.com`

(a) Front view

(b) Rear view

Figure 4.2 – Phoenix (3U) CubeSat (credit: Sarah Rogers, `http://phxcubesat.asu.edu/`)

At present, CubeSats become more and more popular. The number of launched CubeSats rapidly increases and is supposed to increase event more rapidly, as Figures 4.3 and 4.4 show. Graphs are taken from the nanosatellite database[5] [52] and they include also other satellites than CubeSats. Nevertheless, Figure 4.4 depicts that CubeSats account for the majority of considered satellites in the database.

The CubeSat project is a success and CubeSats are currently built not only at universities but also by space agencies, companies and other educational institutions, like high schools. Figure 4.5 shows that two main users are companies and universities.

The number of nanosatellites launched by countries is represented in Figure 4.6 and it can be seen that nanosatellites are built throughout the world.

4.2.1 Mission

Regarding the CubeSat mission, they are primarily used for scientific investigations, most frequently Space Weather and Earth Science [108]. Several CubeSats also serve to test new design and equipment. Some examples of realised or scheduled CubeSat missions are as follows:

— Study the effects of the microgravity environment on biological cultures (GeneSat-1, 2003)[6]
— Detect earthquakes (QuakeSat, 2003)[27]
— Establish a radio connection, download telemetry and receive data from the telescope taking images of the airglow emissions (SwissCube, 2009) [114]
— Test a micro-propulsion system (amorphous hydrogenated Silicon solar cells), a new radio platform,

5. According to [52], the database includes and the term *nanosatellite* implies:
— All CubeSats (0.25U to 27U),
— Nanosatellites (1 kg to 10 kg),
— Picosatellites (100 g to 1 kg),
— PocketQubes, TubeSats, SunCunes and ThinSats
and the database does not include:
— Femtosatellites (10 g to 100 g), chipsats and suborbital launches,
— Satellite in idea or concept phase,
— Data before 1998 (there were at least 21 nanosatellites launches in the 1960s and one in 1997).

6. `https://directory.eoportal.org/web/eoportal/satellite-missions/g/genesat`

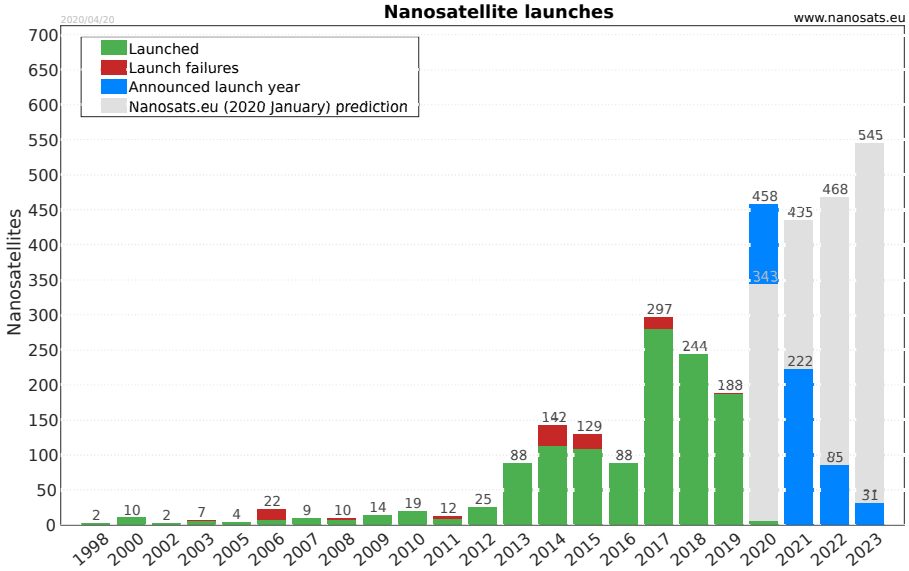


Figure 4.3 – Number of launched nanosatellites per year (As of April 20, 2020; taken from [52])

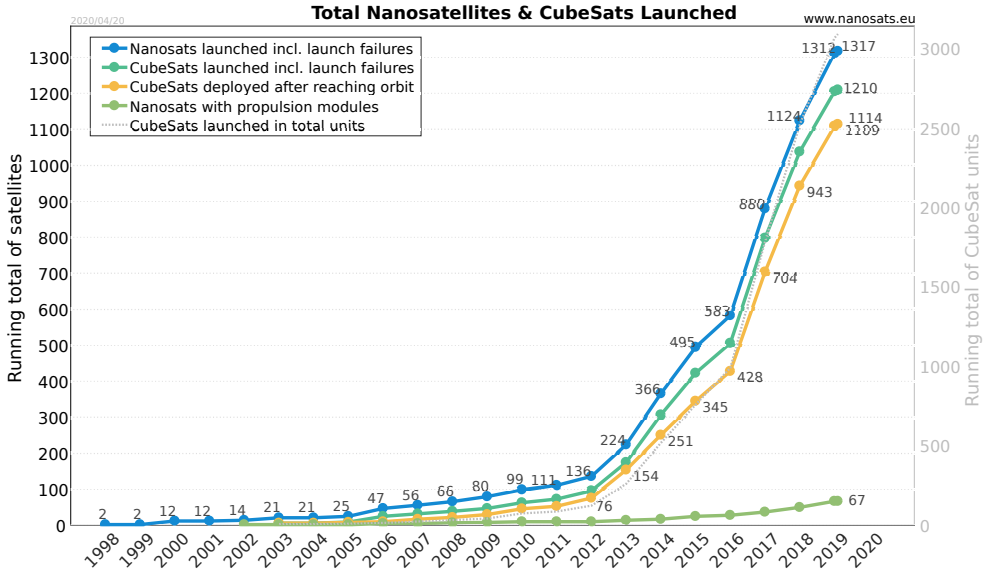


Figure 4.4 – Cumulative sum of launched nanosatellites (As of April 20, 2020; taken from [52])

an agile electrical power system and an active attitude control subsystem [7] (Delfi-n3XT, 2013)
— Test the electric solar wind sail (ESTCUBE-1, 2013) [90]
— Test the piezo motor activity in space, the position identification equipment and data processing algorithms [8] (LituanicaSAT-1, 2014)

7. `https://www.tudelft.nl/en/ae/delfi-space/delfi-n3xt/`
8. `http://www.litsat1.eu/en/`

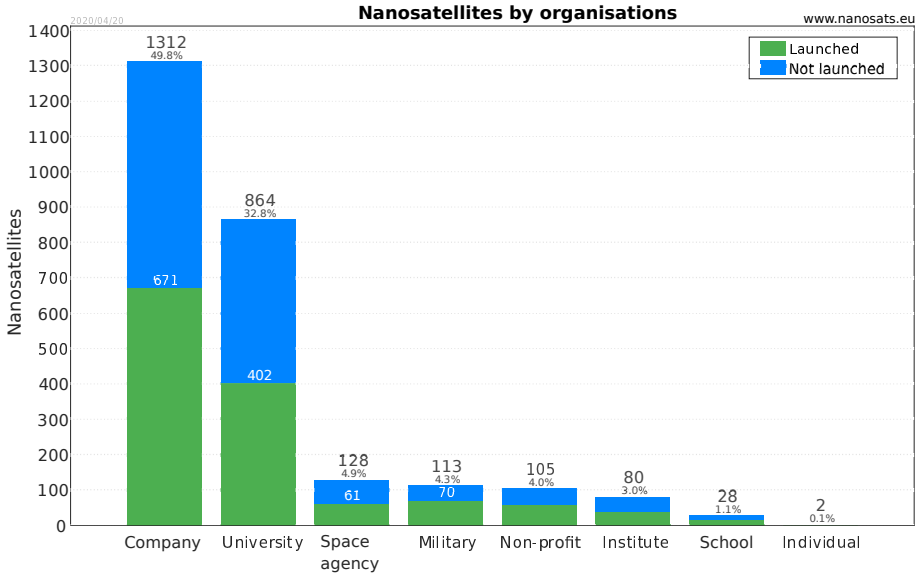


Figure 4.5 – Number of launched satellites by institution (As of April 20, 2020; taken from [52])

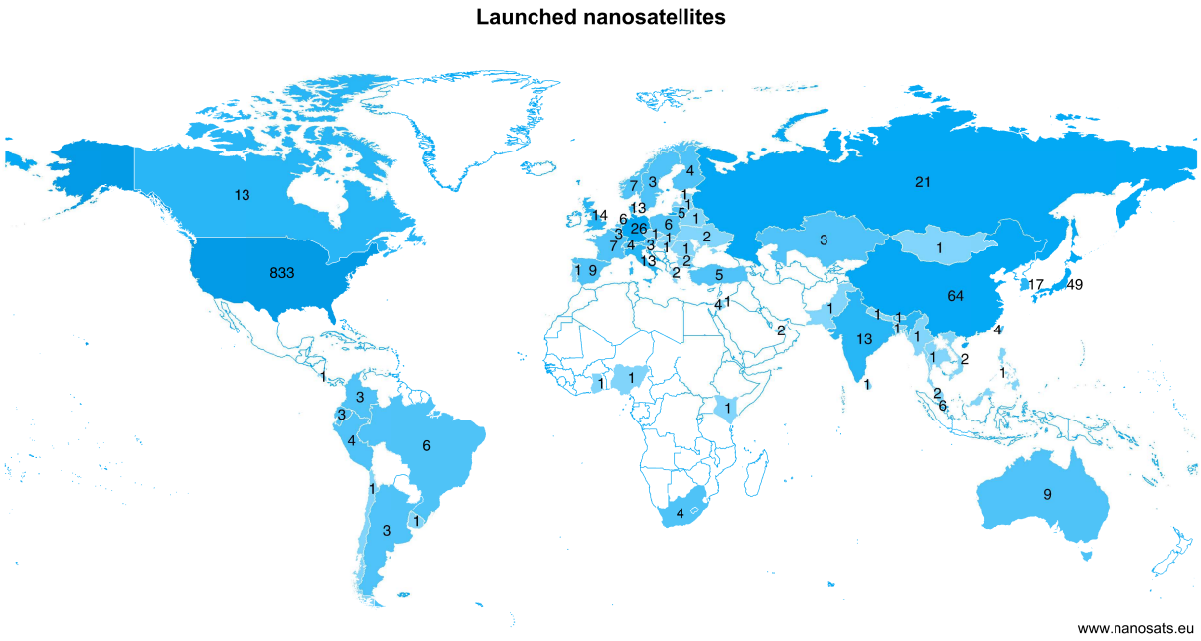


Figure 4.6 – Number of launched satellites by countries (As of April 20, 2020; taken from [52])

— Better understand the solar irradiance energy distribution of solar flare soft X-ray emission and its impact on Earth's ionosphere, thermosphere and mesosphere [9] (MinXSS, 2015)
— Realise an ultra-sail 260 m long and 20 $sq\,m$ solar reflecting film ribbon deployment experiment [10]

9. http://lasp.colorado.edu/home/minxss/science/
10. https://aerospace.illinois.edu/research/research-facilities/laboratory-advanced-space-systems-illinois-lassi

(CubeSail, 2018)
— Test the concept of the new design of the deorbiting sail and test the new design of the sun sensor device (PW-SAT2, 2018) [95]
— Study Urban Heat Islands[11] (UHI) from Low Earth Orbit (LEO) through infrared sensing (PHX SAT, 2019) [13]
— Study atmospheric gravity waves[10] (LAICE, 2019)
— Realise freeze-casting experiments to measure solidification velocity, dendrite and wall width, and particle concentration[10] (SpaceICE, 2020)
— Measure electron density (APSS, 2020) [93, 124]
— Carry out the Earth's imagery[12] (Doves, launched regularly) [84]

Furthermore, researchers intend to use CubeSats also for the following missions:
— Measure water quality using CubeSat hyperspectral imager [11]
— Use constellations of CubeSats to monitor regions after a disaster, such as floods, landslides, earthquakes or fires [127]
— Use CubeSats to provide an alternative global coverage for the Internet of Things (IoT) and Machine-to-machine (M2M) communications [8]

Last but not least, designing and building of CubeSats at universities have also educational purposes. In fact, students working together and putting into practice their knowledge gain rewarding experience.

4.2.2 Systems

CubeSats consist of several systems ensuring correct operations. Since the design can be fully developed by team or can be based on already prefabricated components, each CubeSat is unique. In general, if it is necessary, each system has its own microcontroller (for example Texas Instruments MSP 430F1611 MCU used on board of SwissCube [114]) in order to provide basic housekeeping and parameter configuration, command execution and ensure communication with other systems. The commonly used systems are as follows:

— ***On-Board Computer*** (OBC) or ***Command and Data Handling System*** (CDHS)
This system executes the flight software. Their main functions are (i) to perform scheduling, execution and verification of telecommands, (ii) to store data from housekeeping and telemetry, (iii) to provide a time reference aboard, and (iv) to make computations for other CubeSat systems, for example the attitude determination and control system [40, 114].
Since this system is responsible for the correct CubeSat operation, the choice of its microcontroller is important. Several examples of used microcontrollers are summarised in Table 4.2. It can be seen that most CubeSats are based on the real-time operating system FreeRTOS.
— ***Attitude Determination and Control System*** (ADCS)
This system consists of sensors, such as magnetometers, gyroscopes, sun or temperature sensors, and of magnetorquers used as actuators. The aim is to control the CubeSat attitude, i.e. to determine the position, velocity and orientation [114]. Although the ADCS has its own microcontroller, it is used mainly to read sensors and control actuators. Data processing of the acquired data is generally processed by command and data handling system [40].
This system is not used on board of each CubeSat because not all CubeSats require to direct in one particular direction. Therefore, it is used on board of satellites having for example a camera as a payload.
— ***Electrical Power System*** (EPS)
This system is responsible for electrical power generation, storage and management [100]. The power is harvested from sun using solar panels and stored in batteries. To illustrate the proportion of times in daylight and in eclipse (for a CubeSat located at the altitude of 600 km), a satellite

11. Urban Heat Islands is a phenomenon where cities tend to have warmer air temperatures than the surrounding rural landscapes.
12. `https://www.planet.com/`

passes 63% in the daylight and 37% in the eclipse during one orbital period [14].
The power management is realised by system microcontroller, which checks the battery voltage and current in solar cells and switches on/off current limiters [114]. The microcontroller is also in most cases responsible for choosing a satellite mode of operation (if a CubeSat implements different operating modes).

— ***COMmunication system*** (COM)
The communication system consists of transceiver(s), receiver(s) and antenna(s) and it communicates with ground stations whenever possible. Most CubeSats have only one ground station. The system microcontroller is in general responsible for managing protocols during data transmission [114].
In general, one CubeSat orbit around the Earth takes between 90 and 100 minutes depending on the CubeSat altitude. If known, orbital periods for several CubeSats are indicated in Table 4.2. During one orbital period, a communication between the CubeSat and its ground station lasts from 5 to 10 minutes [93, 114]. A CubeSat flights approximately 15 times round the Earth during 24 hours [14, 114].
Nonetheless, as a CubeSat changes its trajectory throughout the time, it happens that there is no communication at all during one orbit. In [14], the authors simulated the duration of a CubeSat pass over its ground station and they found out that data transmission were realisable only during 6 flights and the time to the next pass can take up to 14 hours and 10 minutes.
To sum up, the higher the altitude, the less orbits around the Earth per day (but variations are minimal), the longer the time spent in the eclipse and the more and longer possible passes [114]. To illustrate these variations, Table 4.1 compares data from three different orbits.

Table 4.1 – Comparison of communication parameters for three orbits [14, 114]

Altitude and beta angle [13]	**400 km and 20°**	**600 km and 0°**	**1000 km and 60°**
Number of orbits around the Earth per day	15	15	14
Number of possible passes over a ground station	5	6	8
Maximum possible duration per pass	8 min	12 min	12 min

As illustrated in Figure 4.7, when a CubeSat orbits the Earth, two main phases can be identified from the scheduling point of view: the *communication phase* and the *no-communication phase*. During the no-communication phase (marked by the red dashed line), there is no communication between the CubeSat and its 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 (highlighted by the green dot-and-dash line), periodic tasks related to communication are executed in addition to the previously mentioned tasks.
The detailed description of data transmission, such as radio frequencies, transmission rates and protocols, between the CubeSat and the ground station(s) is beyond the scope of this thesis.

— ***Payload***
Depending on the mission, a payload can be for example a camera, sensors or tethers. The payload has usually its own microcontroller that is responsible for the payload control and communication between the payload and the command and data handling system. Some examples of several missions to show different types of payload are given in Section 4.2.1.

The systems are then inserted in a structure complying with the CubeSat standards and connected together. Afterwards, solar panels and/or antennas are attached to the structure. Nevertheless, depending on the mission, a CubeSat may not have all these systems. For example, some CubeSats do not need an attitude determination and control system, because their orientation does not influence payload measurements, such as for example APSS CubeSat measuring electron density [93, 124].

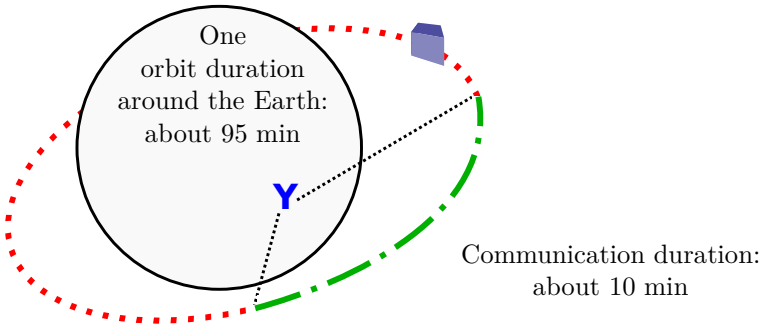


Figure 4.7 – Communication phase (green dot-and-dash line) and no-communication phase (red dashed line)

4.2.3 General Tasks

In general, the tasks that are executed aboard CubeSats can be divided into three categories according to their function.

— ***Housekeeping***
The aim of these tasks is to control all systems. They are carried out by any system microcontroller and they are responsible for (i) receiving and distributing commands to other systems, and (ii) gathering and processing housekeeping and mission data [115].

— ***Payload***
The tasks related to payload are responsible for payload control, data acquisition and data saving.

— ***Communication***
The tasks associated with the communication are in control of gathering housekeeping and payload data, and preparing them to transmit to the ground station. Whenever a communication between a CubeSat and a ground station is possible, they send ready data and receive telecommands from the ground station and treat them.

Table 4.2 – Parameters of several CubeSats (Several data provided by [1, 53])

Name	Size	University/Company	Launch date	Current status	Period (min)	OBC MCU	OS	References
APSS-I	1U	University of Auckland, New Zealand	Dec 30, 2020	Scheduled	?	Delivered by Clyde Space (Cortex-M3 processor)	FreeRTOS	[93] https://space.auckland.ac.nz/ https://apss.space.auckland.ac.nz/
PHX SAT	3U	Arizona State University, USA	Feb 19, 2020	Partly operational	?	Atmel AVR32 (AT32UC3C0512C)	FreeRTOS	[13] http://phxcubesat.asu.edu/
PW-SAT2	2U	Warsaw University of Technology, Poland	Dec 3, 2018	Operational	95.3	(main) STM32F103ZGT6 (reserve) ATXMega128A1	FreeRTOS	[3] https://pw-sat.pl/en/home-page/
RANGE	1.5U	Georgia Tech, USA	Dec 3, 2018	Operational	96.3	Atmel AVR32 MCU	FreeRTOS	[69] http://www.ssdl.gatech.edu/research
CSUNSat1	2U	California State University Northridge and Jet Propulsion Laboratory, USA	May 17, 2017	Reentry May 5, 2019	?	16-bit Microchip dsPIC-33F MCU	Deterministic state machine	[23] http://www.csun.edu/cubesat/
PHOENIX CubeSat	2U	National Cheng Kung University, Taiwan	May 17, 2017	Reentry May 8, 2019	?	32-bit ARM7 RISC CPU	FreeRTOS	[36]
UPSat	2U	University of Patras, Greece	April 18, 2017	Reentry Nov 12, 2018	?	STM32F405 MCU	FreeRTOS	https://upsat.gr/
MinXSS	3U	University of Colorado at Boulder and Laboratory for Atmospheric and Space Physics, USA	Dec 6, 2015	Reentry May 5, 2017	?	16-bit Microchip DSPic32 MCU	RTOS	http://lasp.colorado.edu/home/minxss/ https://eoportal.org/web/eoportal/satellite-missions/content/-/article/minxss#minxss-2
Lituanica-SAT-1	2U	Kaunas University of Technology, Lithuania	Jan 9, 2014	Reentry July 28, 2014	?	(main) 32-bit ARM Cortex M4F CPU (reserve) Arduino 8-bit Atmega2560	FreeRTOS	[141] http://www.litsat1.eu/en/
FUNcube-1	1U	AMSAT-UK, UK	Nov 21, 2013	Operational	97.2	Atmel AT32 MCU	FreeRTOS	[19] https://funcube.org.uk/
Delfi-n3XT	3U	Delft University of Technology, Netherlands	Nov 21, 2013	Contact lost Feb 21, 2014	98.1	Texas Instruments MSP430F1611 MCU (twice)	FreeRTOS	https://www.tudelft.nl/en/ae/delfi-space/delfi-n3xt/ https://directory.eoportal.org/web/eoportal/satellite-missions/d/delfi-n3xt
ArduSat	1U	Spire, former Nanosatisfi (company), USA	Aug 3, 2013	Reentry April 16, 2014	?	(master) Atmel ATmega2561 MCU (nodes) Atmel ATmega328P MCU	?	https://www.spire.com/en https://www.freetronics.com.au/collections/ardusat/products/ardusat-payload-processor-module
ESTCube-1	1U	University of Tartu, Estonia	May 7, 2013	Solar panel degradation May 19, 2015	97.8	ARM Cortex-M3 core (STM32F103) MCU (twice)	FreeRTOS	[90] https://www.estcube.eu/en/home
SwissCube	1U	UNINE/HES-SO/EPFL, Lausanne, Switzerland	Sept 23, 2009	Operational	98.9	32-bit Atmel ARM AT91M55800A processor	?	[114] https://swisscube.epfl.ch/
Compass-1	1U	Aachen University of Applied Sciences, Germany	April 28, 2008	Retired April 14, 2012	96.4	8-bit C8051F123 MCU	?	[129] http://www.raumfahrt.fh-aachen.de/compass-1/home.htm
GeneSat-1	3U	NASA/Santa Clara University, USA	Dec 16, 2006	Reentry Aug 4, 2010	?	Microchip PIC processor	?	https://directory.eoportal.org/web/eoportal/satellite-missions/g/genesat
QuakeSat	3U	Stanford University, USA	June 30, 2003	Battery dysfunction Dec, 2004	101.3	Diamond Systems, Prometheus PC/104 CPU (Motorola PIC 16F628-20P)	Diamond Systems Linux OS	[27] https://www.quakefinder.com/science/about-quakesat/ https://directory.eoportal.org/web/eoportal/satellite-missions/q/quakesat
AAUSAT 1	1U	Aalborg University, Denmark	June 30, 2003	Battery dysfunction Sept 22, 2003	?	16-bit Siemens C161IPI MCU	FreeRTOS (RTX166)	[2] http://www.space.aau.dk/cubesat/

4.3 Space Environment

Space is a harsh environment containing plasma, particle radiation, neutral gas particles, ultraviolet and X-ray radiations, micrometeoroid and orbital debris [89]. From the viewpoint of satellites, they operate in void, under extreme temperature variations and intense accelerations and are subject to space radiation [115]. In this section, we will focus on the radiation because it causes faults in electronic devices. The higher the altitude, the more radiation effects. Although CubeSats are mostly situated at the low Earth orbit (LEO), which is the lowest Earth orbit and located up to 2 000 km of altitude, the radiation should be taken into account.

The space radiation has several sources and varies over time, as well as its effect on electronics [14]. Its sources, depicted in Figure 4.8, are solar wind, solar energetic particles (such as solar flares), galactic cosmic rays, which are high energy particles, and particles trapped in the Earth's magnetic field [22]. Actually, when the radiation approaches the Earth, particles (mainly protons and electrons) are affected by the Earth's magnetic field and form radiation belts called *Van Allen belts*. They are two of them: inner and outer belts and they are located above the LEO. Nevertheless, as the true North does not exactly correspond with the magnetic North, the Earth's magnetic field is asymmetric. This difference causes high concentrations of particles at lower altitudes in the Atlantic near Argentina and Brazil. This phenomenon is called *South Atlantic Anomaly* (SAA). It is located at an altitude between 200 and 800 km over the Earth's surface and it presents a threat to spacecraft passing through [10, 14, 107, 109].

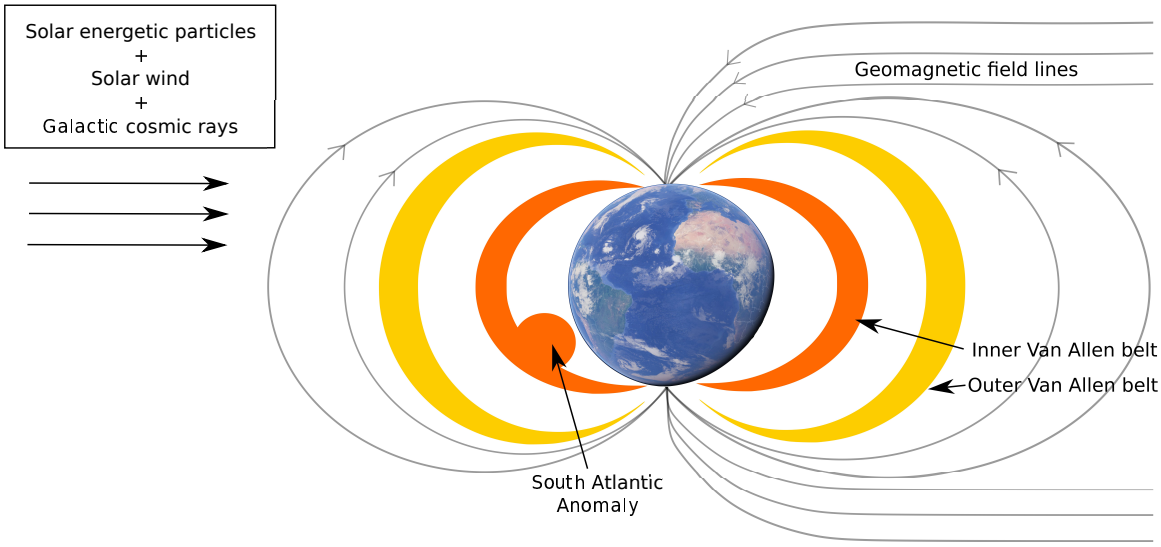


Figure 4.8 – Space environment (Adapted from [10, Figure 2], [22, Slides 4 and 6] and [57, Figure 2.1] and satellite map taken from `https://www.google.fr/maps`)

The radiation effects are generally divided into two categories: (i) long-term, and (ii) transient or single particle effects [89].

The **long-term effects** are mainly due to protons and electrons, which accumulate on electronic components. To evaluate this phenomenon, the metric called *Total Ionising Dose (TID)*, also known as *absorbed radiation dose*, is used. It accounts for the accumulation of ionising dose over time. This metric is habitually expressed in *rad*, where $1\,rad = 0.01\,J/kg$. The material, which is considered, is mentioned in parentheses, e.g $rad(Si)$ for silicon. Nonetheless, the unit of this metric in the international system is gray GY, where $1\,GY = 100\,rads$ [14, 22, 115]. Since materials have different characteristics, the choice of the material is important. For example, it was found in [34] that a GaAs random-access memory (RAM) is more sensitive to lower energy protons than the Si devices.

According to the National Aeronautics and Space Administration NASA [107], satellites and space

vehicles situated at the low Earth orbit and having low beta angle [13] (less than 28°) have a typical dose rate between 100 and 1 000 $rad(Si)/year$. When their inclination is higher (between 20° and 85°), which is the case for the majority of CubeSats, a typical dose rate is between 1 000 and 10 000 $rad(Si)/year$ because of the increased number of trapped electrons.

In [31], the author studied radiation sensitivity of COTS components and it found out that heterogeneous systems on a chip (SoCs) permanently lose their functionality if a TID is higher than 15 $krad$, which shows the importance of the fault protection of CubeSat components.

The space radiation can also cause **transient or single particle effects** that originated by an ion strike. This phenomenon is called *Single Event Effect* (SEE) [89]. It was shown that their occurrences increase for example during a solar flare but their influence also depends on the device [34]. Depending on the effect, the following terms are defined (the list is not exhaustive) [78, 89]:

— *Single Event Upset* (SEU) causes a change of logic state.
— *Single Event Multiple Bit Error* (SEMBE) gives rise to more than one logic state change from one ion.
— *Single Event Transient* (SET) generates a transient current in circuit.
— *Single Event Functional Interrupt* (SEFI) causes that a device enters a mode in which it is no longer performing the designed function.
— *Single Event Latch-up* (SEL) provokes a destructive high current state.
— *Single Event Burnout* (SEB) and *Single Event Gate Rupture* (SEGR) cause a destructive failure of a power transistor.

The aforementioned terms can be then divided into two categories depending on the caused damage [89]:

— *Soft errors*, such as SEU, SEMBE, SEFI or SET, give rise to a temporary faulty state. To return to the normal state, a reset or rewriting is necessary.
— *Hard errors* are destructive because impaired components cannot be used anymore. These errors are due to SEL, SEB or SEGR.

The metric used to measure effects of charged particles is the *Linear Energy Transfer* (LET). It is the rate at which particles deposit the energy into the material and it is a function of the incident energy, particle mass and material density. The unit of this metric is $MeV \cdot cm^2/mg$ [10, 115].

Other metrics, which are commonly used, are *fault rate*, *error rate* and *failure rate*. They respectively account for the number of faults, errors or failures within a time interval. This rate is sometimes calculated for a device or chip during a 24-hour period and the computed value is then normalized to the number of bits [80]. The result is expressed for example in *errors/bit-day*.

In general, it is possible to distinguish three types of components in terms of their robustness against faults: *commercial* (also known as commercial off-the-shelf (COTS)), *radiation tolerant* and *radiation hardened* components [107]. Their characteristics at low Earth orbit are summarised in Table 4.3.

Table 4.3 – Component characteristics at low Earth orbit (altitude < 2 000 km) [107]

Type of components	**Total dose** ($krad$)	**SEU threshold LET** ($MeV \cdot cm^2/mg$)	**SEU error rate** ($errors/bit\text{-}day$)
Commercial components	2 to 10	5	10^{-5}
Radiation tolerant components	20 to 50	20	10^{-7} to 10^{-8}
Radiation hardened components	200 to 1000	80 to 150	10^{-10} to 10^{-12}

The author of [10] devised a radiation tolerant system consisting of one Motorola 7457 processor, two radiation tolerant Actel AX2000 FPGAs and memories protected by error detection and correction

13. The *beta angle* is an angle between the sun and the orbit plane and it determines how long a LEO satellite is exposed to the Sun. It varies from −90° to +90°. The closer to 0° the value, the longer the eclipse [14].

(EDAC). The tests conducted on an orbit of 12 000 km altitude and the inclination of 10 degrees demonstrated that the overall system upset rate was $2.9 \cdot 10^{-4}$ upsets/device/day and the failure rate of the system was approximately $1.5 \cdot 10^{-6}$ per hour (not including radiation induced upsets).

More examples of data related to the fault occurrences have been already presented in Section 1.3. For example, Table 1.5 sums up the failure rates at the International Space Station and Table 1.3 summarises the fault/failures occurrences in space applications.

4.4 Fault Tolerance of CubeSats

Figure 4.9 depicts the present status of launched nanosatellites. Although it can be seen that the majority of launched nanosatellites (687 out of 1317 nanosatellites, i.e. 52.1%) are operational, the number of launched nanosatellites, which are not operational, is high. In general, it is not easy to identify the reason why a nanosatellite did not correctly function and failed, even if we can know for some nanosatellites that the failure occurred during the launch or during the deployment phase. In fact, these phases are most vulnerable to failure.

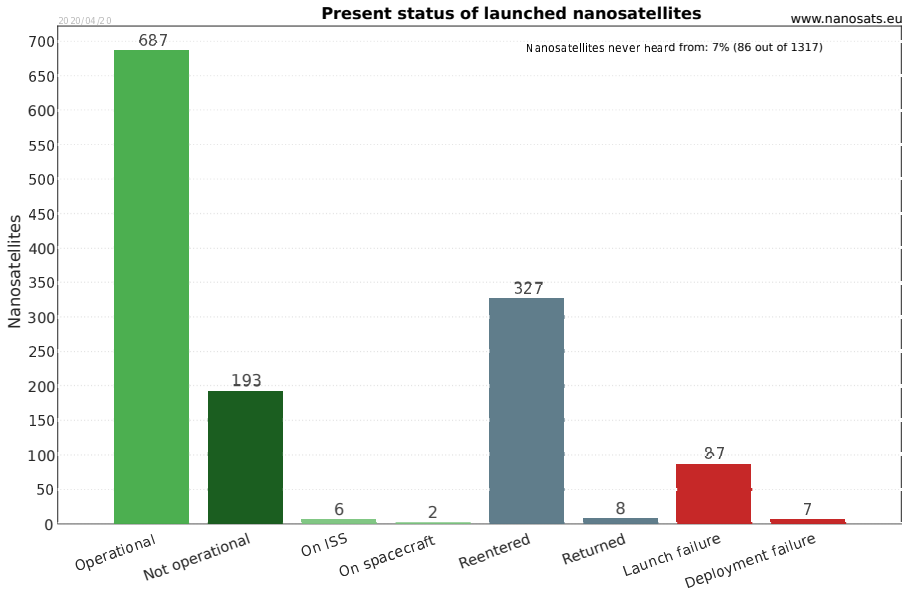


Figure 4.9 – Number of launched nanosatellites and their status (As of April 20, 2020; taken from [52])

In [92], the authors analysed the available satellite data and they found out that 20% of CubeSats are dead on arrival (DOA). Actually, nanosatellites experienced higher infant mortality and DOA rates when compared to other larger satellites. This is mainly due to less testing prior to launch and consequently satellites are launched with undetected errors, which may cause a failure. Based on the analysis of failure data, systematic errors, e.g. failures in design and manufacturing, are the most frequent sources of satellite failure [91]. Moreover, when a CubeSat is transported by a rocket or a launch vehicle to the desired orbit, it must be completely electrically neutral, which means that its batteries must be flat. Once a CubeSat is released in the space, it needs to harvest energy and boot itself [124].

Furthermore, the author of [91] analysed statistical data (mainly based on analysis of 1 584 satellites studied in "Spacecraft Reliability and Multi-State Failures" (2011) by J. H. Saleh and J.-F. Castet) and concluded that, if a satellite is correctly deployed, its projected lifetime (2 or 3 years) will be achieved with probability of more than 90%. Moreover, he stated that, if a failure occurs, 82% of failures is due to software and remaining percents are caused by hardware. Although larger satellites have lower infant

mortality and DOA rates, similar results are obtained but the proportion of software and hardware failures are slightly different.

In order to find a rationale of aforementioned proportions of software and hardware failures, it is necessary to realise that the software complexity, commonly measured in source lines of codes (SLOC) is exponentially increasing. According to [50, 91], flight software grows by a factor of ten every ten years. For example, in 1969 the Boeing 747 airplane worked with approximately 400 000 SLOC and in 2009 the Boeing 787 airplane had approximately 13 000 000 SLOC. Regarding the military aircraft, while software in the F-4A had roughly 1 000 SLOC in 1960, software in the F-22 had 1 700 000 SLOC in 2000 and software in the F-35 has about 5 700 000 SLOC nowadays.

To compare these values with CubeSats, there were 10 000 SLOC of flight code, of which 3 000 SLOC describing device drivers, aboard QuakeSat in 2003 [27].

The main problem related to the CubeSat reliability is that the fault tolerance is not taken into account. For instance, one of the techniques to make CubeSats more robust is to use redundancy, which is described in Section 4.5. In [54], the authors analysed the use of redundancy on board of 159 CubeSats launched before 2014. The results depicted in Figure 4.10 visualise that 43% of CubeSats did not make use of redundancy at all and only 6% had all systems redundant. This low use can be partly explained by the fact that the backup component requires a space, which is limited on board of the CubeSat.

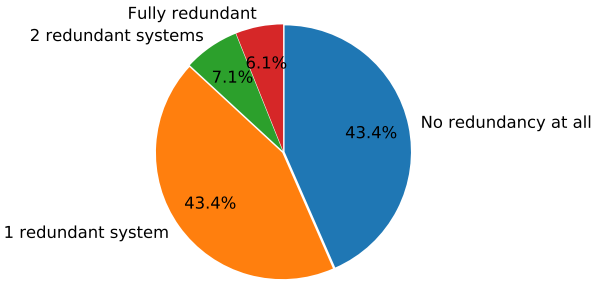


Figure 4.10 – Use of redundancy aboard CubeSats (Adapted from [54, Figure 1])

Last but not least, a survey [55] links the mission success and the use of commercial-off-the-self (COTS) processors: the more COTS components, the lower probability of mission success. This results can be easily explained by the fact that COTS components are not hardened and they are thereby more vulnerable to faults in the harsh space environment.

4.5 Fault Detection, Isolation and Recovery Aboard CubeSats

As it was presented in Sections 4.3 and 4.4, CubeSats do not correctly function due to both design flaw and hostile space environment. In order to overcome this problem and make satellites more robust, there are various approaches described in literature that can fall into the following categories [14]:

— ***Anticipation***
  This category of the fault detection is based on the anticipation of fault occurrence. This can be achieved thanks to one of the following techniques:
  — *Fault detection mechanism* [29, 36] is aimed at supervising tasks (using for example additional dedicated tasks) to detect an anomalous behaviour.
  — *Scan of important parameters* [36], such as processor characteristics, can reveal abnormal values indicating that something is wrong. For instance, the monitoring of power consumption can detect SEL and prevent burnout [30].

14. This classification was compiled by author of this thesis.

- *Analysis of error reports* [36, 90] allows users to find reasons for failure occurrences and consequently to upgrade the system to be able to tolerate such faults in the future. A rectification can be made directly in the code or a correction can be defined in a library describing how to recover from failures [36].
- *Kalman filtering* is sometimes used as an advanced technique to predict the satellite condition in the future and detect a failure [29]. Due to its complexity, it is not commonly put into practice on board of CubeSats. A rare example of its implementation is ESTCube-1 [90].
- *Turning off processors* improves the system reliability because switched off components are less vulnerable to faults. This technique is not often used on board of CubeSats but it is employed aboard the Hubble Space Telescope. In fact, when it passes through the South Atlantic Anomaly (described in Section 4.3), it does not carry out any observation [57].
- Last but not least, *prior tests before launch* [29] make possible to detect faults and correct them before the beginning of the mission.

— ***Redundancy***

In order to fulfil a mission, it is recommended to have a backup component able to take over the duties of the first component if it becomes faulty [3, 29, 36, 141]. In literature, two types are distinguished: *hot redundancy* and *cold redundancy* [90]. The backup components using the former type are always turned on so that they are ready to immediately provide results in the case the first component is faulty. When the latter type is put into practice, the backup components are turned off and switched on once the first component is unable to correctly function. Nevertheless, due to space constraints aboard CubeSats, it may not be possible to make use of this fault tolerant technique.

As an example, we consider the use of redundancy for on-board computer, which represents an important part of CubeSat because its malfunction jeopardises the mission. In [3], the authors differentiated three cases:

- If *only one microcontroller* is used, there is no redundancy and, in case of dysfunction, the mission is aborted. This solution is often chosen and was used for example on board of AAUSat, Compass-1 Picosatellite, PHOENIX CubeSat or SWEET CubeSat.
- In order to use a redundancy, *two identical and independent microcontrollers* can be put on board. A drawback of this approach is that two identical microcontrollers are subject to space environment, in particular to ionizing radiation, at similar rate. Therefore, it is likely that the backup component will malfunction soon after the first one. This solution was chosen by team designing the ESTCube-1 [90].
- To overcome the problem related to the degradation at similar rate, *two different independent microcontrollers* can be applied. The main microcontroller ensures smooth operation of the mission. If it becomes defective, a backup microcontroller takes over its function. It is less powerful but able to execute vital tasks so that the mission could continue despite degraded functioning. Since the second component is less advanced, it degrades slowly than the main one. This approach was chosen for PW-Sat2 [3].

— ***Watchdog timer and reset/reboot***

If a system malfunctions, for example it is latched up or frozen, one of the possibilities is to reboot or reset the system. A commonly used technique is the *watchdog timer* [3, 29, 30, 36, 38, 100, 114, 141]. In principle, if time is up, for example if a watchdog does not receive a heartbeat from a processor within the defined time [27], software is reset and/or a satellite is rebooted.

Although this solution is usually used to recover from a faulty state, it can be periodically employed to avoid a fault occurrence. For instance, QuakeSat rebooted the system every two weeks [27].

— ***Checkpointing***

This method consists in periodic saving of data during the execution. If a fault occurs, while a task is executing, its execution is restarted from the last checkpoint or from scratch if no checkpoint exists. On board of satellites, this technique was put into practice for example in [56].

— ***Remote control***

Since there is a regular communication between a CubeSat and ground stations, the CubeSat transmits reports on its current status. These reports can be analysed and operators can send telecommands in order to configure or upgrade on-board software during the mission [29, 38, 90].

— ***Safe mode***
If a fault is detected, the command and data handling system can decide that the CubeSat switches its state machine to the *safe mode* [29, 38, 114, 141]. To quit this state, a system needs to recover from fault, e.g. thanks to the reboot or telecommands sent by operators.

— ***Data protection***
Faults can also occur when transmitting data or when accessing memory, i.e. during reading or storing data. Techniques for data protection are usually based on information redundancy, which adds check bits to data in order to verify the correctness [85]. The commonly used techniques are the checksum [38], cyclic redundancy checks (CRC) [36] and Hamming codes [30, 38].
It is also possible to triplicate the memory [23, 90] or interconnect all systems together to avoid a failure in communication [90].

— ***Radiation hardened components***
A possibility of dealing with radiation is to use components designed for the harsh space environment. Nevertheless, they are rather expensive and not all teams can afford them [10, 19]. Table 4.3 compares radiation hardened and commercial off-the-shelf components in terms of the capability to withstand the radiation.

— ***Shielding***
Shielding is a simple method to protect a component or the whole system against radiation [10, 30]. This method cannot reflect all particles but it reduces their number. Since each component (even a COTS one) has a certain level of the radiation sensitivity, the aim of shielding is not to exceed this level. However, shielding is not applicable to all types of radiation. It is an efficient protection against TID but useless against single event effects.
In [31], the authors studied fault tolerance of satellite on-board computers based on COTS components. They found out that an aluminium shielding of 1.5 mm is sufficient for a small satellite to correctly function during 3 years, which generally corresponds to the end of a typical mission length.
Since CubeSats have strict weight constraints, the main drawback of this method is the increase in the total satellite mass.

4.6 Summary

This chapter presented CubeSats and the harsh space environment where they operate. Although they become more and more popular thanks to the standardisation of components and rather affordable budget, their missions are not always fulfilled. One of the main causes is that CubeSats are not so robust as they should be to withstand faults caused for example by radiation. No matter the reason of the lack of fault tolerance (for example budget or space constraints), a solution to make CubeSats more robust against faults is one of the main achievements of this thesis and is presented in the next chapter.

Chapter 5

Online Fault Tolerant Scheduling Algorithms for CubeSats

The preceding chapter introduced small satellites called CubeSats, which become more and more popular and are built not only at universities but also by companies and space agencies [52]. It was shown that since they operate in the harsh space environment and the fault tolerance is not always considered due to for example budget or time constraints, they are vulnerable to faults.

To support CubeSat teams, this chapter comes up with a solution to make CubeSats more fault tolerant. After the introduction of the idea, we present our system, task and fault models. Then the algorithms and experimental framework are described. Subsequently we carry out the analysis and discuss the results.

We present two no-energy-aware algorithms, OneOff and OneOff&Cyclic, and then one energy-aware algorithm called OneOffEnergy.

5.1 Our Idea

Our aim is to provide CubeSats with the fault tolerance. As there are several systems aboard CubeSats and most of them have its own processor, we present a solution gathering all processors on one board. This modification will not only reduce space and weight and optimise the energy consumption but also improve the system resilience. First, a shielding against radiation will be easier to put into practice [30, 31], as described in Section 4.5. 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 in current CubeSats) and each processor can execute any task. Although this design is not typical nowadays, it has been successfully realised for example on board of ArduSat, which has 17 processors on one board [58].

Once all processors are gathered on one board, we intend to use the proposed scheduling algorithms dealing with all tasks (regardless of 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 (COTS) processors, which are not necessarily designed to be used in space applications and therefore more vulnerable to faults than radiation hardened processors. It was reported [55] that the more COTS components in a CubeSat, the lower the probability of its mission success.

5.2 No-Energy-Aware Algorithms

5.2.1 System, Fault and Task Models

Table 5.1 summarises notations and definitions used in our research related to CubeSats.

The studied system consists of P interconnected identical processors. Although the system is composed of homogeneous processors sharing the same memory, it would be possible to extend it to a system composed of heterogeneous processors, like in [155]. The system handles all tasks on board of the CubeSat.

Table 5.1 – Notations and definitions

Notation	Definition
a_i	Arrival time of task t_i
ϕ_i	Phase of task t_i
T_i	Period of task t_i
et_i	Execution time of task t_i
d_i	Deadline of task t_i
tt_i	Task type of task t_i
α	Multiple of et_i to define the size of PC scheduling window
PC_i	Primary copy of task t_i
BC_i	Backup copy of task t_i
xC_i	PC or BC of task t_i
$start(xC_i)$	Start of the execution of PC_i or BC_i
$end(xC_i)$	End of the execution of PC_i or BC_i
S	Simple task
D	Double task

These tasks are mostly related to housekeeping (e.g. sensor measurements), communication with ground station and storing or reading data from the memory.

The task model distinguishes aperiodic and periodic tasks. An *aperiodic task*, depicted in Figure 5.1, 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 5.2, 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 respectively before the deadline or the beginning of the next period.

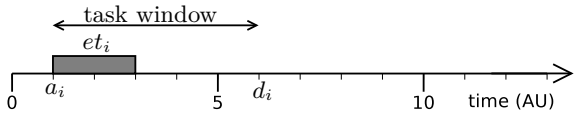


Figure 5.1 – Model of aperiodic task t_i

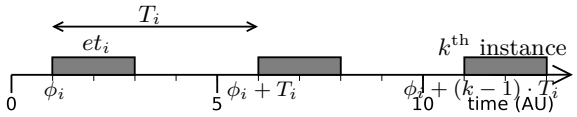


Figure 5.2 – Model of periodic task τ_i [33]

The fault model 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 distinguish 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 and only if a primary copy is faulty, the corresponding backup copy is scheduled. The algorithm consequently schedules backup copies only when it is necessary, which avoids waste of resources.

Simple tasks have only one primary copy because a fault is detected by timeout, no received acknowledgment or failure of data checks. By contrast, a fault detection for *double tasks* requires the execution of two primary copies [1] 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.

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.

1. 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.

Therefore, using Graham's notation [66] described in Section 1.1, the studied problem is defined as:

$$P; m \mid n = k;\ online\ r_j;\ d_j = d;\ p_j = p \mid \text{(minimise the rejection rate)}$$

which means that k independent jobs/tasks (characterised by release time r_j, processing time p_j and deadline d_j) arrive online on a system consisting of m parallel identical machines and are scheduled to minimise the rejection rate.

5.2.2 Presentation of Algorithms

This section describes two algorithms meant for online global scheduling on a multiprocessor system. 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. In order not to increase the algorithm run-time, we analyse several underlying ordering policies at the beginning to finally choose one policy minimising the rejection rate. 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 (respectively labelled $PC_{i,1}$ and $PC_{i,2}$) in a fault-free environment.

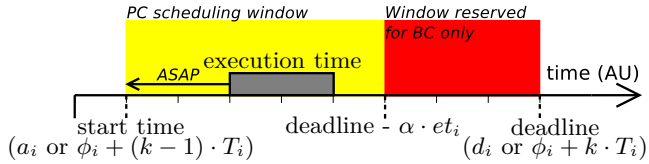


Figure 5.3 – Principle of scheduling task copies

As Figure 5.3 shows, all primary copies are scheduled as soon as possible to avoid idle processors just after 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 $\alpha \geqslant 1$). We consider without loss of generality that $\alpha = 1$.

When the algorithm finds out that a primary copy was 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. The proposed algorithms guarantee that, if any primary copy is faulty, its corresponding backup copy can be always scheduled and executed. Actually, from the scheduling point of view, the backup copies of all accepted tasks can always be scheduled and executed. Nevertheless, it may happen that a backup copy is impacted by a fault too. Therefore, we distinguish two metrics, described in Section 5.2.3.2 to evaluate both the system schedulability by means of the rejection rate and the number of correctly executed tasks by the system throughput.

Regarding the processor allocation, we call the *slot*, a time interval within the 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. This processor allocation strategy corresponds to the one we proposed for the primary/backup approach and called the *first found solution search slot by slot*. It is described in Section 3.1.1.2. Although

the principle is the same for both cases, the selection of processor on which the search for a slot starts is different. In the context of CubeSats, processors are ordered according the first available time. An example is depicted in Figure 5.4, where xC_i stands for primary or backup copy of task t_i.

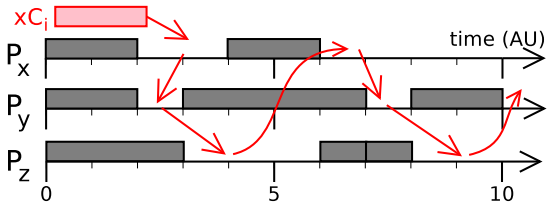


Figure 5.4 – Principle of the algorithm search for a free slot on processors

5.2.2.1 Mathematical Programming Formulation

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

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

subject to

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

The purpose of the objective function of our scheduling problem consists in maximising the number of accepted tasks, which is equivalent to minimising of the task rejection rate. The first constraint is related to the PC scheduling window depicted in Figure 5.3 considering $\alpha = 1$. The second constraint forbids task copies of the same task to be scheduled on the same processor. The third one accounts 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.

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

The online algorithm scheduling arriving tasks as the aperiodic ones is called OneOff in this thesis. This name was derived from "One task off" meaning that at each scheduling trigger at least one aperiodic task is scheduled.

When OneOff is used, all tasks are considered as aperiodic, which means that each instance of periodic task is transformed into an aperiodic task. In such a case, 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.

The main steps of OneOff are summarised in Algorithm 13.

Algorithm 13 Online algorithm scheduling all tasks as aperiodic tasks (OneOff)

Input: Mapping and scheduling of already scheduled tasks, (task t_i, fault)
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                                  ▷ a processor is idle and a task arrives and/or a 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:        for each ordering policy do
14:            Order the task queue
15:            for each task in the task queue do
16:                Map and schedule its task copies (PC(s) or BC)
17:        Choose the ordering policy whose schedule has the lowest rejection rate
18:        if a scheduled task copy starts at time t then
19:            Commit this task copy
20:        else
21:            Nothing to do
```

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 schedule is not necessary and task copies are committed using the already existent schedule (Lines 2-6).

Otherwise (Lines 7-21), new task copies (PC(s) for new task and/or a BC for task impacted by fault) are added to the task queue and the algorithm removes all task copies that have not yet started their execution. Afterwards, each ordering policy orders tasks in the queue and the algorithm searches for a new schedule. Finally (Lines 17-21), the schedule minimising the rejection rate is chosen and the task copies starting at time t are committed.

To avoid ordering the task queue for each policy and to reduce the algorithm run-time, our aim is to evaluate several policies (listed at the beginning of Section 5.2.2) and their combination to finally choose the one minimising the rejection rate. Consequently, while at the beginning of the result analysis several ordering policies are considered, only one policy, which were chosen based on its performances, is studied later.

The complexity of one search for a schedule where N denotes the number of tasks in the task queue and P is the number of processors is as follows. The complexity to order a task queue is $O(N \log(N))$ and the one to add a task in an already ordered queue is $O(N)$. 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 reads:

$$O(N + P \cdot N \cdot (\#\ task\ copies) + 1) \tag{5.1}$$

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 is as follows:

$$(\text{maximum theoretical \# of scheduling searches}) = (\text{\# tasks at the input}) + (\text{\# task copies}) \tag{5.2}$$

In order to reduce this number, we present a method making use of a buffer, which is a commonly used technique in scheduling [46, 81]. 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

$$d_i - \text{current time} - et_i \leqslant K \cdot et_i \qquad \text{where } K \in \mathbb{N} \tag{5.3}$$

otherwise, it is called *large*.

The principle of the method is illustrated in Figure 5.5. The highlighted background shows the part that was added to the baseline version. To enter it, the algorithm checks the slack using Formula 5.3 where $K = \beta$ and the current time equals the task arrival. 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 scheduled.

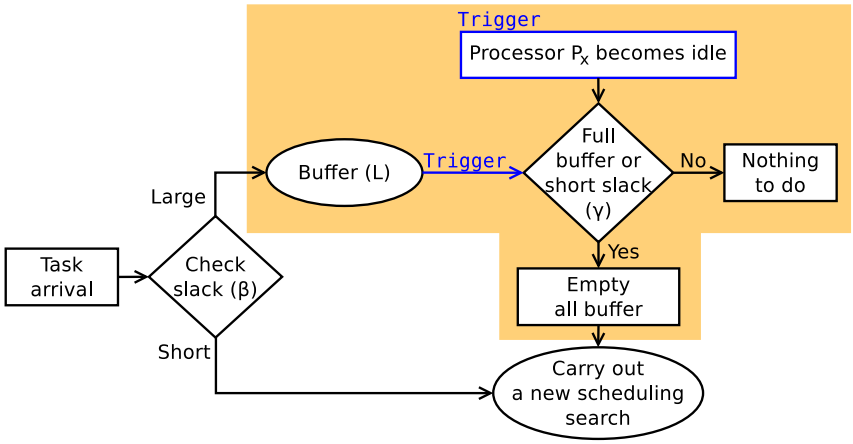


Figure 5.5 – Principle of the method to reduce the number of scheduling searches

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

The online algorithm scheduling arriving tasks as the aperiodic or periodic ones is called OneOff&Cyclic. Its name is derived from OneOff. The term Cyclic is appended because the algorithm is also able to deal with the periodic tasks and to repeat an already determined schedule of one hyperperiod (HT) until a new scheduling trigger occurs.

OneOff&Cyclic is consequently aware that there are not only aperiodic tasks but also periodic ones and there are two task sets: one for periodic tasks and one for aperiodic ones.

The main steps of OneOff&Cyclic are summed up in Algorithm 14.

Algorithm 14 Online algorithm scheduling all tasks as periodic or aperiodic tasks (OneOff&Cyclic)

Input: Mapping and scheduling of already scheduled tasks, (task t_i, fault)
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 ▷ a 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:        for each ordering policy do
16:            Order the task queues
17:            for each task in the task queue of aperiodic tasks do
18:                Map and schedule its task copies (PC(s) or BC)
19:            for each task in the task queue of periodic tasks do
20:                Map and schedule its task copies (PC(s) or BC)
21:        Choose the ordering policy whose schedule has the lowest rejection rate
22:        if a scheduled task copy starts at time t then
23:            Commit this task copy
24:        else
25:            Nothing to do
```

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[2] of periodic task(s), or (iv) a fault during task execution.

In the case a processor becomes idle (Case (i)), a new search for schedule is not carried out and task copies are committed using the already determined 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) occurs.

Otherwise (Lines 7-25), 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 16-20), task sets are ordered and the algorithm schedules aperiodic tasks and periodic ones. In general, since there are none or only a few tasks in a set of aperiodic tasks (accounting mainly for interrupts), the choice of ordering policy is not important compared to the one for periodic tasks. Finally (Lines 21-25), the schedule minimising the rejection rate is chosen and the task copies starting at time t are committed.

Again, our goal is to assess several ordering policies (listed at the beginning of Section 5.2.2) and their combination to select the one minimising the rejection rate in order to avoid ordering the task queue several times and reduce the algorithm run-time. Consequently, while at the beginning of the result analysis several ordering policies for periodic tasks are considered, only one policy is studied later.

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

2. A possibility to add or 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 5.2.3.

complexity is as reads:

$$O(N_{aper} + P \cdot N_{aper} \cdot (\#\ task\ copies) + N_{per} + P \cdot N_{per} \cdot (\#\ task\ copies) + 1) \tag{5.4}$$

5.2.3 Experimental Framework

In Section 4.2.2, it was mentioned that two main phases during the CubeSat orbit can be identified from the scheduling point of view: the *communication* and *no-communication phases*, as depicted in Figure 4.7. Recall that CubeSats mainly execute periodic tasks related to e.g. telemetry, reading/storing data or checks. Interrupts due to an unexpected or asynchronous event are considered as aperiodic tasks. These tasks are executed during both communication and no-communication phases. In addition, during the communication phase, CubeSats deal with tasks associated with the communication. Since these tasks are periodically repeated when a communication takes place, but are not present during the no-communication phase, they are called *sporadic*.

Next, we describe our simulation scenario and define metrics employed to analyse the presented algorithms.

5.2.3.1 Simulation Scenario

The data exploited in our experimental framework are based on real CubeSat data provided by the Auckland Program for Space Systems (APSS) [3] and by the Space Systems Design Lab (SSDL) [4]. These data were gathered by their functionality and generalised in order to generate more data for simulations. They are respectively called Scenario APSS and Scenario RANGE and summarised in Tables 5.2 and 5.3, where U denotes a uniform distribution, the arbitrary time unit is $1\,ms$ and one hyperperiod is the least common multiple of task periods.

Table 5.2 – Set of tasks for Scenario APSS

Periodic tasks

Function	**Task type** tt_i	**Phase** ϕ_i	**Period** T_i	**Execution time** et_i	**# tasks**
Communication	D	U(0; T)	500 ms	U(1 ms; 10 ms)	2
Reading data	S	U(0; T)	1 000 ms	U(100 ms; 500 ms)	10
Telemetry	D	U(0; T)	5 000 ms	U(1 ms; 10 ms)	2
Storing data	S	U(0; T)	10 000 ms	U(100 ms; 500 ms)	7
Readings	D	U(0; T)	60 000 ms	U(1 ms; 10 ms)	2

Sporadic tasks related to communication

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

Aperiodic tasks

Function	**Task type** tt_i	**Arrival time** a_i	**Execution time** et_i	**# tasks**
Interrupts	D	U(0; 100 000 ms)	U(1 ms; 10 ms)	1

In order to further analyse the algorithm performances (see Section 5.2.4), we also modified Scenario APSS. This scenario is called Scenario APSS-modified and its data are summed up in Table 5.4. Its tasks are the same as for Scenario APSS but the periods of 500 ms were prolonged to 1 000 ms and periods longer than 5 000 ms were shortened to 5 000 ms. The number of tasks, whose periods were modified, per period were computed pro-rata and rounded in order to have similar system load and proportion of simple and double tasks as for Scenario APSS.

The number of task copies per hyperperiod in a fault-free environment for each aforementioned scenarios is given in Table 5.5.

3. `https://space.auckland.ac.nz/auckland-program-for-space-systems-apss/`
4. `http://www.ssdl.gatech.edu/`

Table 5.3 – Set of tasks for Scenario RANGE

Periodic tasks

Function	Task type tt_i	Phase ϕ_i	Period T_i	Execution time et_i	# tasks
Kalman filter	D	U(0; T)	100 ms	U(1 ms; 30 ms)	1
Attitude control	D	U(0; T)	100 ms	U(10 ms; 30 ms)	1
Sensor polling	D	U(0; T)	100 ms	U(1 ms; 5 ms)	5
Telemetry gathering	S	U(0; T)	20 000 ms	U(100 ms; 500 ms)	1
Telemetry beaconing	S	U(0; T)	30 000 ms	U(10 ms; 100 ms)	2
Self-check	D	U(0; T)	30 000 ms	U(1 ms; 10 ms)	5

Sporadic tasks related to communication

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

Aperiodic tasks

Function	Task type tt_i	Arrival time a_i	Exec. time et_i	# tasks
Interrupts, GPS	D	U(0; 10 000 ms)	U(1 ms; 50 ms)	10

Table 5.4 – Set of tasks for Scenario APSS-modified

Periodic tasks

Function	Task type	Phase ϕ_i	Period T_i	Execution time et_i	# tasks
Communication	D	U(0; T)	1 000 ms	U(1 ms; 10 ms)	4
Reading data	S	U(0; T)	1 000 ms	U(100 ms; 500 ms)	10
Telemetry	D	U(0; T)	5 000 ms	U(1 ms; 10 ms)	2
Storing data	S	U(0; T)	5 000 ms	U(100 ms; 500 ms)	3
Readings	D	U(0; T)	5 000 ms	U(1 ms; 10 ms)	1

Sporadic tasks related to communication

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

Aperiodic tasks

Function	Task type	Arrival time a_i	Execution time et_i	# tasks
Interrupts	D	U(0; 5 000 ms)	U(1 ms; 10 ms)	1

Table 5.5 – Number of task copies for three scenarios

Scenario	# task copies per hyperperiod in a fault-free environment	
	Communication phase	No-communication phase
APSS	6696	1176
RANGE	9647	8447
APSS-modified	561	101

To model dynamic aspect, although task sets are defined for simulations in advance, they are not known to algorithms until discrete simulation time has equalled arrival time (for aperiodic tasks) or phase (for periodic and sporadic tasks).

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

To compare our results, we defined the mathematical programming formulation of our problem as described in Section 5.2.2.1 and carried out resolutions in CPLEX optimiser [5] using the same data set.

Since tasks dynamically arrive, a real-time aspect needs to be modelled. Actually, it is not possible to resolve the scheduling problem only once because CPLEX optimiser would know all task characteristics in advance and it would be an offline instead of an online scheduling. Similarly to resolutions in CPLEX

5. https://www.ibm.com/analytics/cplex-optimizer

solver described in Section 3.1.2.1, at each scheduling trigger, the main function updates task 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 related to the dynamic task arrival of aperiodic tasks, only results for OneOff&Cyclic were obtained. Actually, OneOff have many scheduling triggers, which make resolutions unfeasible within the reasonable time.

Fault Generation

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 [118]. We therefore 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 the algorithm performances 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.

Several curves are based on less than 20 simulations due to computational constraints. During the communication phase for systems consisting of 9 or 10 processors and OneOff&Cyclic at $1 \cdot 10^{-3}$ fault/ms, there are no results due to hard computational time constraints. Actually, even 150 hours, which corresponds to the maximum possible computation time available on the computing grid on which simulations were carried out, was not sufficient for one simulation to be finished.

5.2.3.2 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 *processor load* is also studied to evaluate the processor utilisation.

To analyse the algorithm run-time, we use the following metrics. 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.

5.2.4 Results

In this section, we first estimate the theoretical processor load and the proportion of simple and double tasks in each scenarios. Then, we compare the rejection rate, analyse the number of scheduling searches and scheduling times. Finally, we evaluate the algorithm performances in the presence of faults.

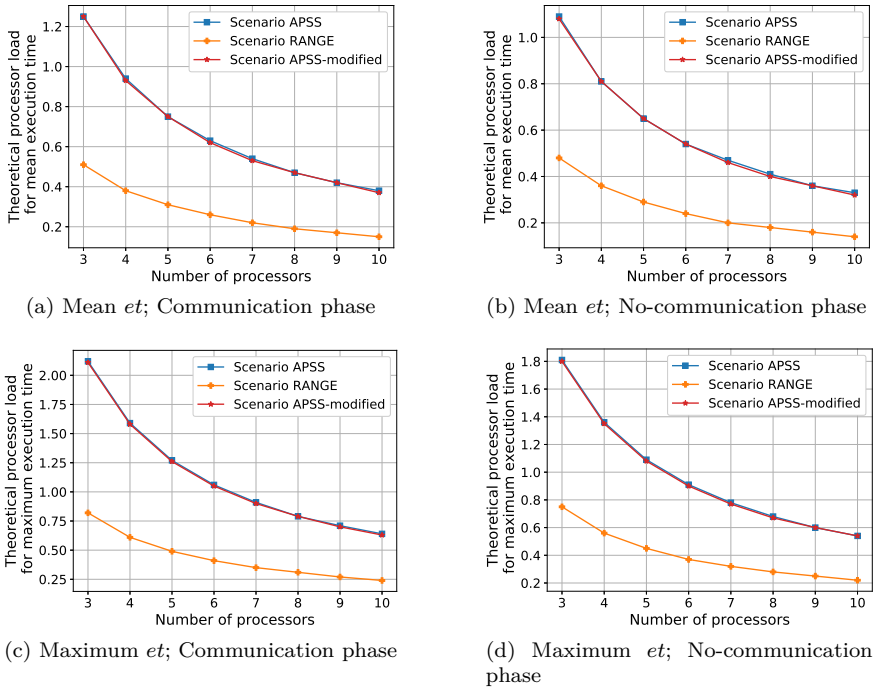
5.2.4.1 Theoretical Processor Load

In this section, we focus on the processor load and task proportions for each scenario in a fault-free environment.

Based on Tables 5.2, 5.3 and 5.4, we compute the theoretical processor load when considering both maximum and mean execution times of each task. The results for three scenarios are depicted in Figures 5.6 representing such processor loads respectively for both communication phases as a function of the number of processors.

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 other two 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 Figures 5.7. It can be observed that during the communication phase the percentage of double tasks for Scenarios APSS



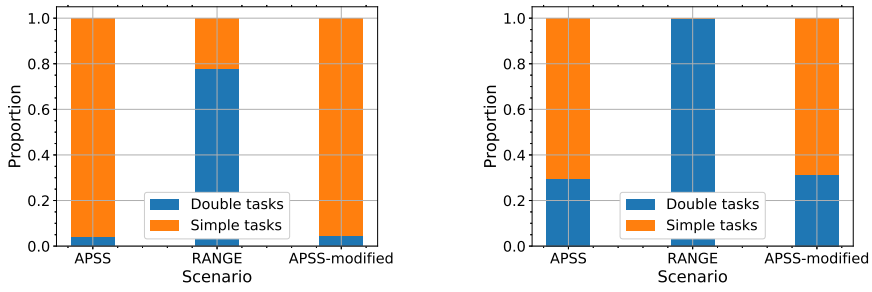
(a) Mean *et*; Communication phase

(b) Mean *et*; No-communication phase

(c) Maximum *et*; Communication phase

(d) Maximum *et*; No-communication phase

Figure 5.6 – Theoretical processor load when considering maximum and mean execution times (*et*) of each task



(a) Communication phase

(b) No-communication phase

Figure 5.7 – Proportion of simple and double tasks

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.

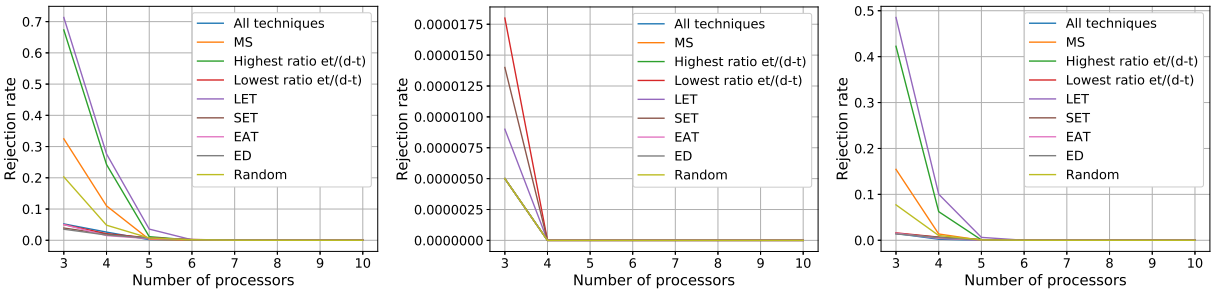
To conclude, our experimental framework makes use of two very different sets of scenarios. On the one hand, Scenarios APSS and APSS-modified have high system load and high proportion of simple tasks compared to double tasks. On the other hand, Scenario RANGE mainly contains double tasks and has lower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

5.2.4.2 Rejection Rate of OneOff and OneOff&Cyclic

We analyse the performances of OneOff and OneOff&Cyclic in terms of the rejection rate. We compare different ordering policies, listed in Section 5.2.2, for three scenarios in order to choose which

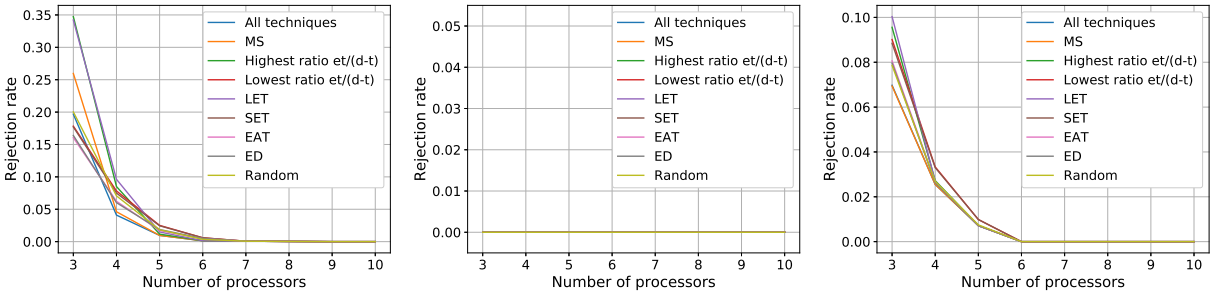
ordering policy is the best in terms of the rejection rate. When an ordering policy is mentioned in the legend it means that it is exclusively used by the algorithm and no other ordering policy is considered, whereas "All techniques" signifies that all ordering policies were tested to find a schedule.

Analysis of OneOff We compare different scenarios when OneOff is implemented. Figures 5.8 and 5.9 respectively show the rejection rate of three scenarios for both communication and no-communication phases as a function of the number of processors. First of all, it can be seen that Scenario RANGE has almost no rejection rate during the communication phase and none rejection rate during the no-communication phase. This is due to the task data set, which has rather low system load as it was aforementioned.



(a) Scenario APSS (b) Scenario RANGE (c) Scenario APSS-modified

Figure 5.8 – Rejection rate as a function of the number of processors (OneOff; communication phase)

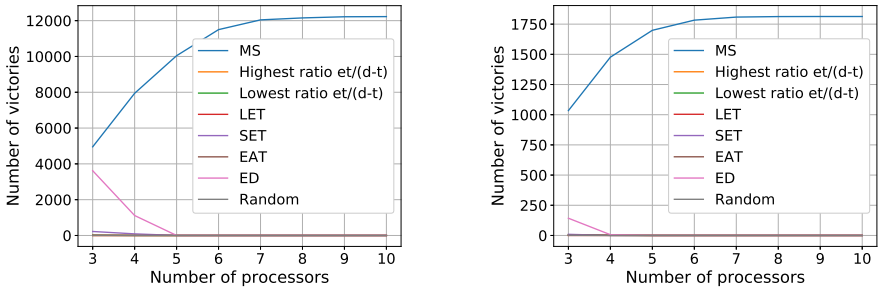


(a) Scenario APSS (b) Scenario RANGE (c) Scenario APSS-modified

Figure 5.9 – Rejection rate as a function of the number of processors (OneOff; no-communication phase)

The "Earliest Deadline" or "Earliest Arrival Time" techniques overall reject the least tasks and "All techniques" does not achieve the lowest rejection rate all the time. To further evaluate the method "All techniques", Figures 5.10 depicts the number of victories among all tested ordering policies for both communication phases. When several ordering policies achieve the same rejection rate, the algorithm chooses the schedule delivered by the first ordering policy in the list. As it can be seen in Figures 5.10, this is often the case when the system consists of more than five processors. In addition, when the number of processors is low, the schedule delivered by the "Earliest Deadline" is chosen, even though this ordering policy is penultimate in the list of tested ordering policies. Consequently, the method "All techniques" will no longer be considered because its performances do not excel and it increases the algorithm run-time since several ordering policies need to be tested, as stated in Algorithm 13 (Line 13).

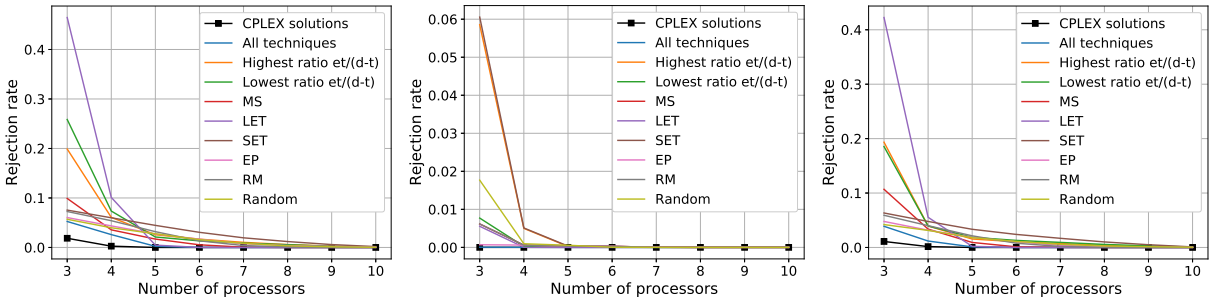
Analysis of OneOff&Cyclic We contrast different scenarios when OneOff&Cyclic is used. Figures 5.11 and 5.12 respectively depict the rejection rate of three scenarios for both communication and



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

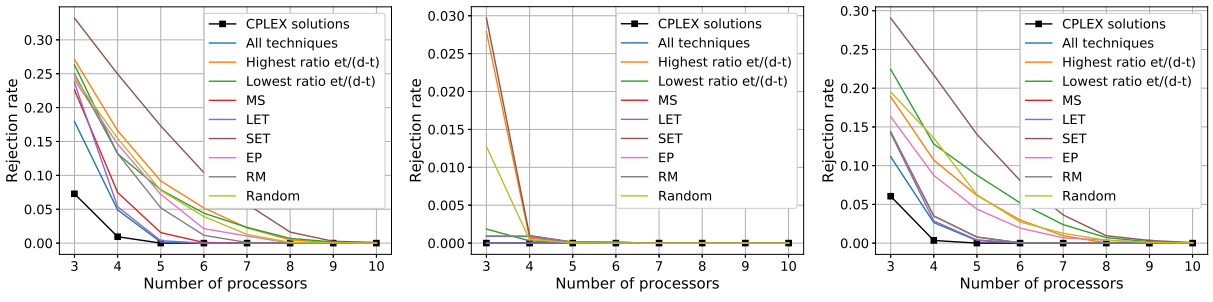
Figure 5.10 – Number of victories for "All techniques" method as a function of the number of processors (OneOff; Scenario APSS)

no-communication phases as a function of the number of processors. These figures depict not only the studied ordering policies and their combination, but also a curve plotting the optimal solution provided by CPLEX solver that is based on the mathematical programming formulation defined in Section 5.2.2.1. 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 the optimal solution.



(a) Scenario APSS (b) Scenario RANGE (c) Scenario APSS-modified

Figure 5.11 – Rejection rate as a function of the number of processors (OneOff&Cyclic; communication phase)



(a) Scenario APSS (b) Scenario RANGE (c) Scenario APSS-modified

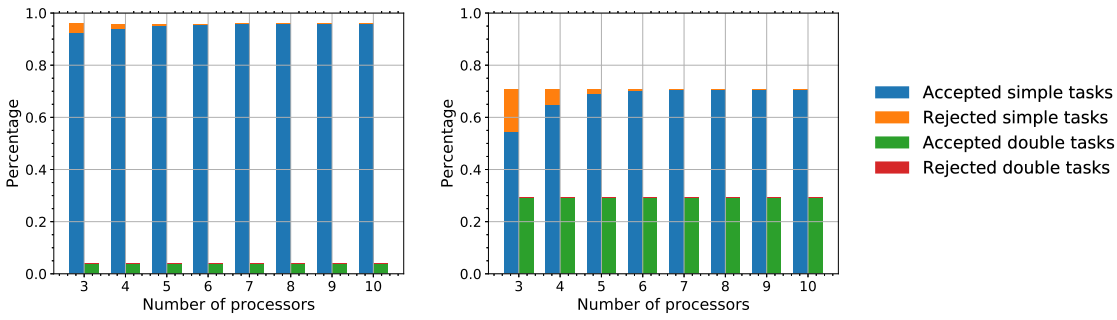
Figure 5.12 – Rejection rate as a function of the number of processors (OneOff&Cyclic; no-communication phase)

Again, Scenario RANGE has several times lower rejection rate than other two scenarios because of

lower system load. Furthermore, it can be seen that it is not so straightforward to determine one ordering policy, which performs well for all scenarios. Although the method "All techniques", testing all ordering policies reject the least tasks, it will not be considered any more due to longer algorithm run-time. Thus, a reasonable choice is the "Minimum Slack" or "Earliest Phase" techniques during the communication phase and the "Minimum Slack" or "Longest Execution Time" during the no-communication phase. Altogether, the "Minimum Slack" ordering policy perform well regardless of the type of phase. Nevertheless, the rejection rate of OneOff&Cyclic is in general higher than the one of OneOff.

Comparison of Different Scenarios The performances of a given ordering policy are influenced by the system load and 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 of rejection rates 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, Figures 5.13 show the proportion of simple and double tasks against the rejection rate for Scenario APSS as a function of the number of processors when OneOff using the "Earliest Deadline" policy is put into practice.



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

Figure 5.13 – 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)

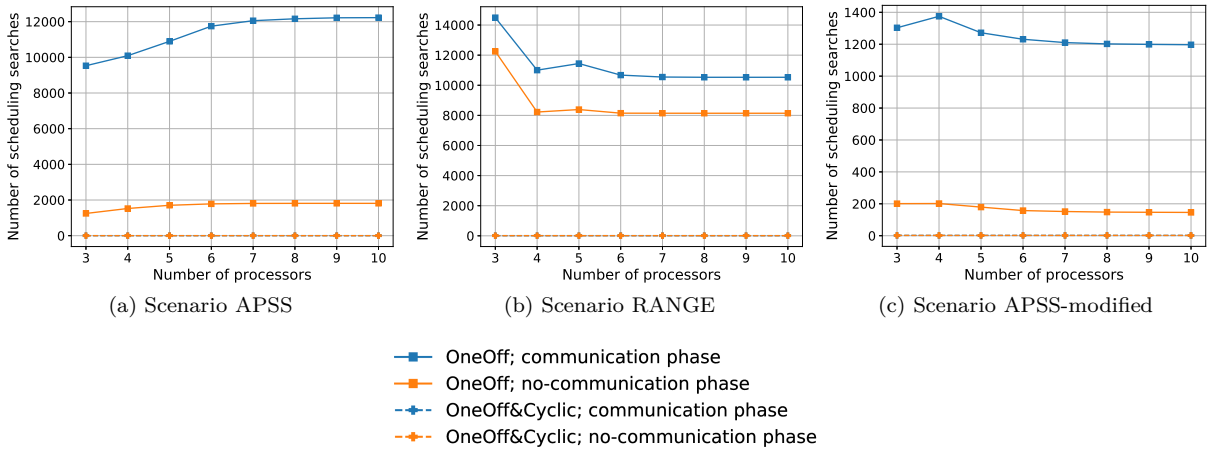
In order not to oversize the system, the analysis of the rejection rate (regardless of the scenario) also shows that it is useless to consider more than six processors. In fact, when an ordering policy is well chosen, no task is rejected.

5.2.4.3 Number of Scheduling Searches

In this section, we compare the number of scheduling searches, i.e. how many times a search for a new schedule was carried out. We consider the "Earliest Deadline" ordering policy when analysing OneOff and the "Minimum Slack" policy for OneOff&Cyclic because it was shown in Section 5.2.4.2 that they achieve the best results in terms of the number of rejected tasks. Figures 5.14 depict the number of scheduling searches for three CubeSat scenarios for both studied algorithm, OneOff and OneOff&Cyclic.

Figures 5.14 point out that the number of scheduling searches of OneOff is significantly higher when compared to OneOff&Cyclic.

As defined in Formula 5.2, the former algorithm (OneOff) has the number of scheduling searches at most equal to the sum of the number of tasks to be scheduled and the number of task copies. In general, the number of scheduling searches is lower than the maximum theoretical value and approximately equals the number of tasks at the input when there are more processors in the system because every task activates



(a) Scenario APSS (b) Scenario RANGE (c) Scenario APSS-modified

Figure 5.14 – Number of scheduling searches as a function of the number of processors [6]

a scheduling trigger. We remind the reader that the number of task copies per hyperperiod is available in Table 5.5. The distance between the curves representing data respectively for the communication and no-communication phases depends on scenario, especially Scenario RANGE differs from Scenarios APSS and APSS-modified as described in Section 5.2.4.2.

The latter algorithm (OneOff&Cyclic) has only two searches for Scenarios APSS and APSS-modified and eleven searches for Scenario RANGE (no matter whether there is a communication or not), as it can be foreseen from Tables 5.2, 5.3 and 5.4. This difference is due to the number of scheduling triggers. While each instance of periodic task does not trigger a new scheduling search for OneOff&Cyclic, it does for OneOff. Thus, it may seem that this algorithm is more useful to avoid high number of scheduling searches than OneOff. Nevertheless, the scheduling time of one scheduling search plays also an important role and it is evaluated in Section 5.2.4.4.

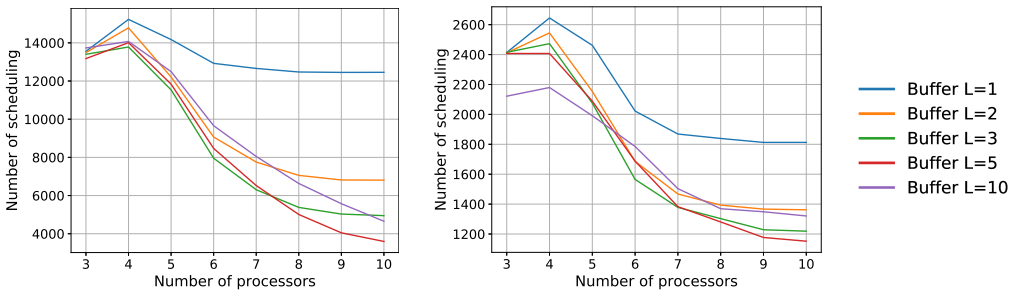
Evaluation of the Method to Reduce the Number of Scheduling Searches for OneOff In Section 5.2.2.2, we presented a method to reduce the number of scheduling searches, which is now assessed in terms of the rejection rate and number of scheduling searches. Figures 5.15 and 5.16 respectively represent these two metrics for Scenario APSS during both communication phases. 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.

Figures 5.15 show 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 respectively 30% and 17% for the communication and no-communication phases. Moreover, when the value of L is high, e.g. 10, the number of scheduling searches is not necessarily lower because the slack of a task in the buffer becomes short and a new search is triggered.

Regarding the rejection rate, presented in Figure 5.16, the longer the buffer, the more tasks rejected. Actually, when a task is put into the buffer, processors may be idle while it is in the buffer. Later, they may not be able to accommodate all tasks, which need to be scheduled.

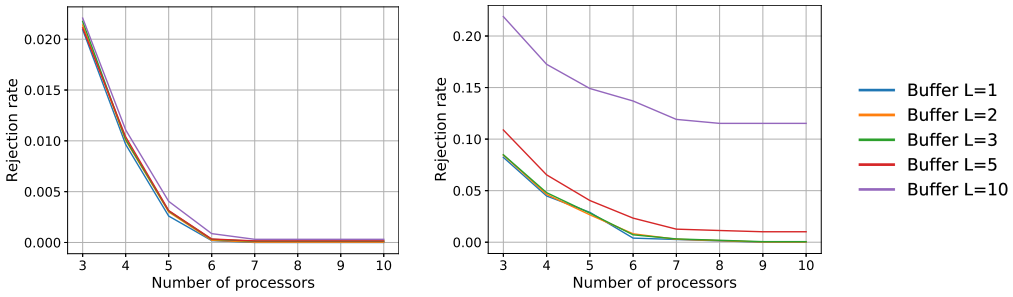
Next, the detailed analysis was carried out in order to find values of the buffer length L and the slack constants β and γ. We found out that these values mainly depend on an application. In general, if the buffer is shorter, there are more scheduling searches because the buffer cannot accommodate more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

6. OneOff&Cyclic has the same number of scheduling searches for both communication phases. Their curves are consequently overlapping.



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

Figure 5.15 – Number of scheduling searches as a function of the number of processors (OneOff; Scenario APSS)



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

Figure 5.16 – Rejection rate as a function of the number of processors (OneOff; Scenario APSS)

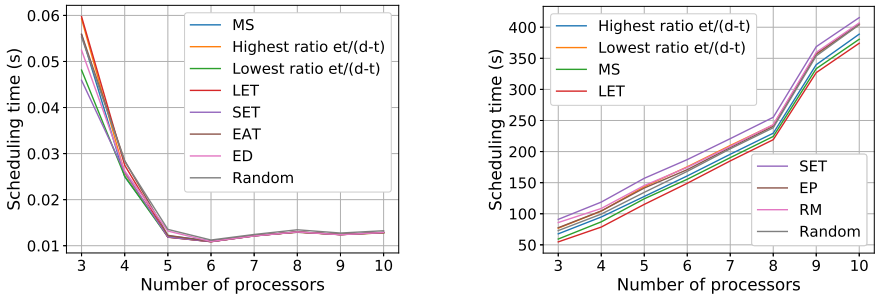
Although the idea to set a limitation on the number of scheduling searches seems interesting, it presents several major drawbacks. First and foremost, such a limitation increases the rejection rate, which is the metric we want to minimise. Besides, when setting the values of β and γ, the algorithm is not general any more because the choice of the values would be probably application-dependent. Last but not least, this method does not maximise the processor utilisation. Consequently, this method of reducing the number of scheduling searches will no longer be considered in this thesis.

5.2.4.4 Scheduling Time

In this section, we compare the scheduling time of OneOff and OneOff&Cyclic for three different scenarios. The policy "All techniques" is not considered because its scheduling time would be the sum of times elapsed by all tested ordering policies, which makes this policy the worst from the viewpoint of the scheduling time.

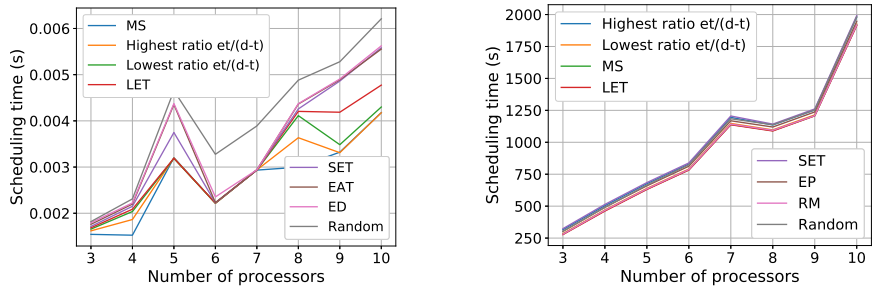
Figures 5.17 represent the scheduling time of **Scenario APSS** respectively for OneOff and OneOff&Cyclic during the no-communication phase as a function of the number of processors. The scheduling time during the communication phase are qualitatively similar to the ones in Figures 5.17 but approximately 4 times longer for OneOff&Cyclic and 2 times longer 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, which achieve the lowest scheduling time for OneOff, are the "Shortest Execution Time", "Lowest ratio of $et/(d-t)$" and "Earliest Deadline". Regarding 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 Mono-



(a) OneOff (b) OneOff&Cyclic

Figure 5.17 – Scheduling time as a function of the number of processors (Scenario APSS; no-communication phase)



(a) OneOff (b) OneOff&Cyclic

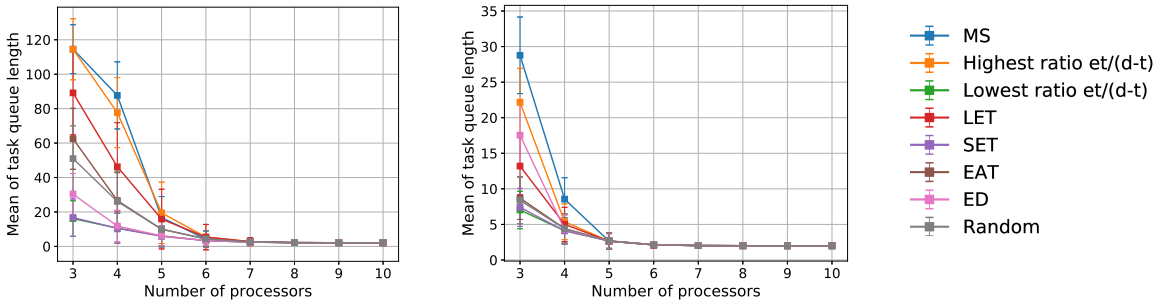
Figure 5.18 – Scheduling time as a function of the number of processors (Scenario RANGE; no-communication phase)

tonic" techniques as the worst ones in terms of the scheduling time. To demonstrate the gap, we consider a 3-processor system: the "Shortest Execution Time" technique needs 536 s during the communication phase and 90 s during the no-communication phase, which is roughly double than the "Longest Execution Time" technique requiring 260 s during the communication phase and 55 s during the no-communication phase

The scheduling times of **Scenario RANGE** respectively for OneOff and OneOff&Cyclic during the no-communication phase as a function of the number of processors are depicted in Figures 5.18. The scheduling times during the communication phase are the same to the ones in Figures 5.18, except that OneOff&Cyclic approximately requires additional 150 s. The best ordering techniques in this case are as follows: the "Minimum Slack" and "Highest ratio of $et/(d-t)$" for OneOff and the "Longest Execution Time" and "Minimum Slack" techniques for OneOff&Cyclic.

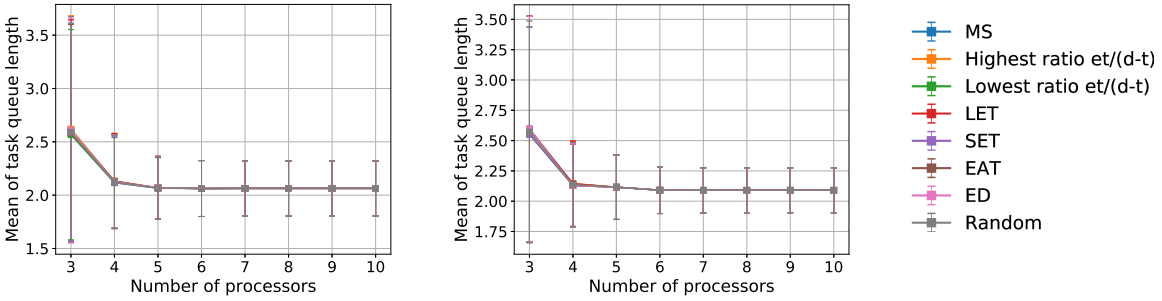
The scheduling time is related to the algorithm complexity, which is defined in Sections 5.2.2.2 and 5.2.2.3 for OneOff and OneOff&Cyclic, respectively. One of the terms accounting for the complexity is the number of tasks in the task queue. To show the trend of the task queue length, Figures 5.19 depict the mean value of task queue length with standard deviations respectively during both communication phases for OneOff and Scenarios APSS and RANGE. We notice that the higher the number of processors, the shorter the task queue and that the number of tasks in the queue depends on the system load. While the ordering policies for Scenario APSS have significant differences in the number of tasks in the task queue when a system has a low number of processors, the ones for Scenario RANGE do not differ because Scenario RANGE has lower system load than Scenario APSS, as shown in Figures 5.6.

Consequently, the scheduling time of OneOff for Scenario APSS decreases with the higher number



(a) Scenario APSS; Communication phase

(b) Scenario APSS; No-communication phase



(d) Scenario RANGE; Communication phase

(e) Scenario RANGE; No-communication phase

Figure 5.19 – Mean value of task queue length with standard deviations as a function of the number of processors (OneOff)

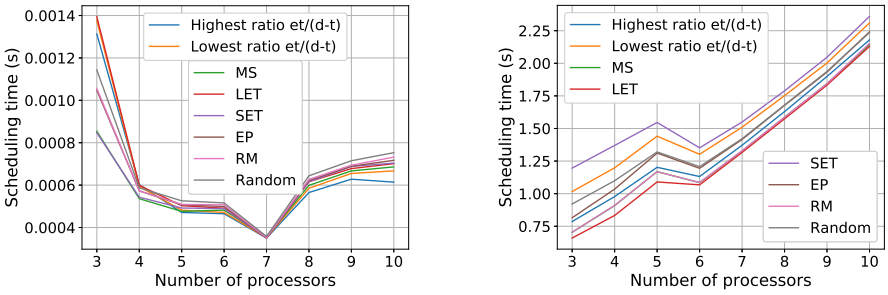
of processors owing to shorter task queue because there are more scheduling triggers. Regarding the scheduling time of OneOff&Cyclic, it is longer when the number of processors increases even though the number of tasks is almost constant (the set of periodic tasks remains the same for a given phase and there is only one (Scenario APSS) or ten (Scenario RANGE) arrivals of aperiodic tasks). The increase is due to more possibilities to be tested when a system has more processors.

Nonetheless, as the results are based on simulations (because real experiments are not easily feasible), scheduling times in our experiments do 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 times will be shorter.

Last but not least, the scheduling time of OneOff&Cyclic is roughly 5 orders of magnitude greater than the one of OneOff. This huge gap is mainly caused by the significant difference in task periods: between 500 ms and 60 000 ms. To better evaluate this impact on scheduling time, we modified Scenario APSS to Scenario APSS-modified, as described in Section 5.2.3.1.

Figures 5.20 represent the scheduling time of **Scenario APSS-modified** respectively for OneOff and OneOff&Cyclic during the no-communication phase as a function of the number of processors. The trend of scheduling times during the communication phase is similar to the ones in Figures 5.20 and the values are multiplied 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).

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 task periods accelerates the scheduling time. We therefore suggest to teams building CubeSats to avoid tasks with very short and very long periods to be scheduled together.

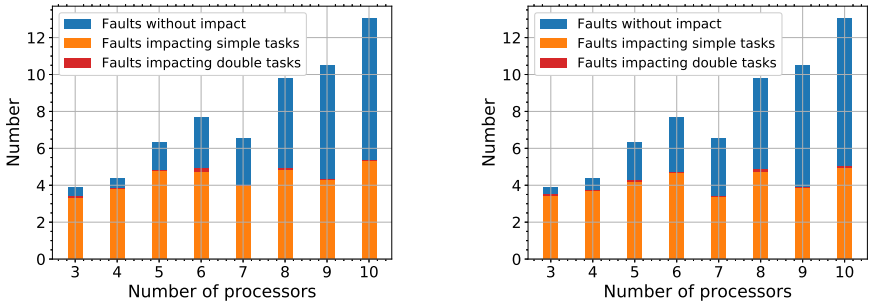


(a) OneOff (b) OneOff&Cyclic

Figure 5.20 – Scheduling time as a function of the number of processors (Scenario APSS-modified; no-communication phase)

5.2.4.5 Simulations with Fault Injection

In this section, we evaluate the fault tolerance of both algorithms for Scenario APSS. We chose this scenario because it is based on real data (and not on a modified version) and it has rather high system load when compared to Scenario RANGE. We consider the "Earliest Deadline" policy for OneOff and the "Minimum Slack" policy for OneOff&Cyclic.



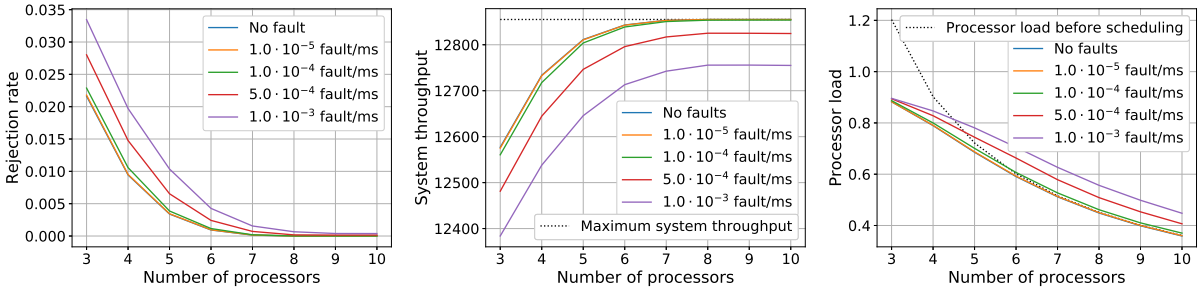
(a) Communication phase (b) No-communication phase

Figure 5.21 – Total number of faults (injected with fault rate $1 \cdot 10^{-5}$ fault/ms) against the number of processors (OneOff; Scenario APSS)

Figures 5.21 depict the total number of faults against the number of processors, while the total number is the sum of the faults without impact, faults impacting simple tasks and faults impacting double tasks. The fault were injected with fault rate $1 \cdot 10^{-5}$ fault/ms, which corresponds to the worst estimated fault rate in the real space environment [118]. 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 they consequently may not be integers.

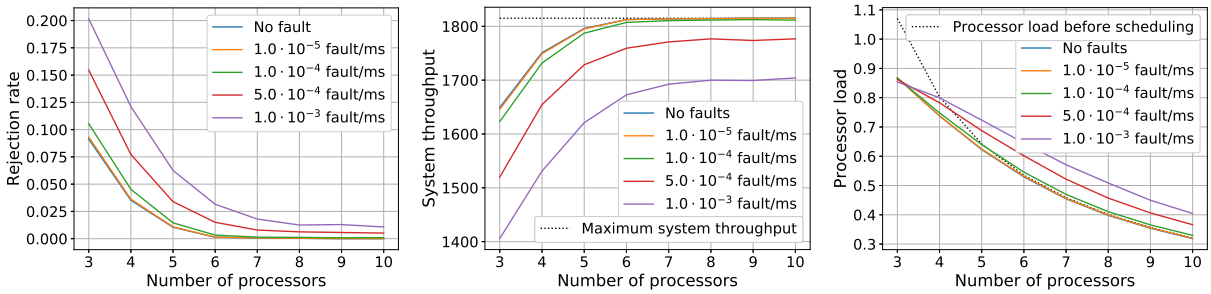
The number of impacted tasks remains almost constant and there is no significant difference between two algorithms nor between communication phases. Furthermore, double tasks are rarely impacted, which is due to their shorter execution time compared with simple tasks. We also studied other fault rates and, as expected, the higher the fault rate, the more faults. Nonetheless, the proportion of impacted simple and double tasks remains the same.

Figures 5.22 and 5.23 respectively depict the rejection rate, system throughput and processor load for both communication phases as a function of the number of processors. Qualitatively similar results were obtained for OneOff&Cyclic. The figures representing the system throughput include a black dashed line corresponding to the case when no task is rejected and all tasks are correctly executed. Regarding the figures plotting the processor load, they also show a black dashed line, which denotes the maximum



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

Figure 5.22 – System metrics at different fault injection rates as a function of the number of processors (OneOff; Scenario APSS; communication phase)



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

Figure 5.23 – System metrics at different fault injection rates as a function of the number of processors (OneOff; Scenario APSS; no-communication phase)

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, 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 characterises the schedulability as described in Section 5.2.3.2, i.e. if a fault occurs during a PC execution, the corresponding backup copy is scheduled. Nevertheless, it may happen that a backup copy is impacted by a fault too. In this case, such a task does not contribute to the system throughput because it was not correctly executed.

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. Furthermore, 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 [118]). The same conclusions were made for other two scenarios (RANGE and APSS-modified) as well.

Although only transient faults were studied, the CubeSat performances 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 5.1. 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 random non-permanent hardware faults in each core during non-bursty period is $10^{-4}/h$ [118].

5.3 Energy-Aware Algorithm

The preceding section presented two algorithms for reduction in the rejection rate that are applicable to CubeSats. It was shown that OneOff, i.e. an online algorithm scheduling arriving tasks as the aperiodic ones, achieves better results in terms of the rejection rate and scheduling time than OneOff&Cyclic, which is an online algorithm scheduling arriving tasks as the aperiodic or periodic ones. Therefore, we consider onwards OneOff only. In this section, this algorithm is enhanced in the sense it also takes into consideration energy constraints because energy is of major interest for satellite applications.

5.3.1 System, Fault and Task Models

While the fault model is exactly the same as described in Section 5.2.1, the system and task models slightly differ.

Since OneOff is put into practice, all tasks are considered as aperiodic. Every task is characterised by the arrival time a_i, execution time et_i, deadline d_i, task type tt_i and task priority tp_i. The last attribute takes on three possible values: *high* (H), *middle* (M) and *low* (L), and it characterises the balance between the task importance and energy consumption. For instance, the tasks related to CubeSat housekeeping have higher priority than the tasks associated with payload. Similarly, while it is appropriate to send a signal that a CubeSat is still operating to the ground station during the communication phase in the case it experiences the energy shortage, it may not be necessary to transmit all reports.

Since there are three task priorities, the system distinguishes three operating modes: *normal* (N), *safe* (S) and *critical* (C). Inspired by [14], the system chooses a mode according to the current battery capacity. Table 5.6 associates each mode with the battery capacity and tasks that are authorised to be executed.

Table 5.6 – System operating modes

Mode	**Battery capacity**	**Executed tasks**		
		Tasks having $tp_i = H$	Tasks having $tp_i = M$	Tasks having $tp_i = L$
Normal	50% – 100%	✓	✓	✓
Safe	20% – 50%	✓	✓	-
Critical	0% – 20%	✓	-	-

We consider that the system regularly harvests energy, e.g. from the sun, stored in a battery and consumes it to power processors. Although the dynamic voltage and frequency scaling may be available when executing tasks, we do not make use of it because it does not improve the reliability, as discussed in Section 1.5. All tasks are thereby always executed at the maximum processor frequency.

Without loss of generality, we take an example of STM32F103 processor based on ARM 32-bit Cortex-M3 CPU because it is commonly used on board of CubeSats, as shown in Table 4.2. Its characteristics for the maximum ($72\,MHz$) and minimum ($125\,kHz$) frequencies and four operating modes are summarised in Table 5.7.

Table 5.7 – Several characteristics of STM32F103 processor

Operating mode	I_{DD}	V_{DD}	$P_{STM32F103}$	**Wakeup time**
Run ($72\,MHz$)	$52.5\,mA$	$3.3\,V$	$173\,mW$	$0\,\mu s$
Run ($125\,kHz$)	$1.4\,mA$	$3.3\,V$	$4.6\,mW$	$0\,\mu s$
Sleep ($72\,MHz$)	$32.5\,mA$	$3.3\,V$	$107\,mW$	$1.8\,\mu s$
Sleep ($125\,kHz$)	$1.35\,mA$	$3.3\,V$	$4.5\,mW$	$1.8\,\mu s$
Stop	$38.7\,\mu A$	$3.3\,V$	$0.13\,mW$	$5.4\,\mu s$
Standby	$2.5\,\mu A$	$3.3\,V$	$0.0083\,mW$	$50\,\mu s$

In order to save energy, we take advantage of various processor operating modes. While the tasks are executed at the maximum frequency ($72\,MHz$), the processor executes in Run mode at the minimum processor frequency ($125\,kHz$) if there is no task executing on a processor. On the one hand, since Sleep and Stop modes exhibit wakeup times and only negligibly lower power consumption, they are not implemented. On the other hand, Standby mode consumes much less energy comparing with Run mode and that is why it is applied to the algorithm.

Since the energy stored in the battery varies continually but not abruptly, the modes and consequently system load do not change very often, so that it allows the algorithm to put several processors into Standby mode. The number of processors switched into Standby mode depends on the operating mode and is summed up in Table 5.8. As it can be seen, the system may operate with less than 3 processors during critical mode.

Table 5.8 – Number of processors in Standby mode

Mode	Normal	Safe	Critical
# processors in Standby mode	0	$\lfloor \frac{1}{6}P \rfloor$	$\lfloor \frac{1}{3}P \rfloor$

We consider that the durations of changes of frequencies and modes, as well as the wakeup time from Standby mode ($50\,\mu s$), are negligible when compared to the time unit ($1\,ms$) in our simulation scenario. To avoid task migration in the case a processor is put into Standby mode, a task copy that is already running on the processor is not suspended and the mode is changed after the end of its execution.

Our objective is to minimise the task rejection rate subject to real-time, reliability and energy constraints. This means to maximise the number of tasks being correctly executed before deadline without depleting all system energy even if a fault occurs.

5.3.2 Presentation of Algorithm

The algorithm taking into account energy constraints is an improved version of OneOff, i.e. the algorithm scheduling all arriving tasks as the aperiodic ones, introduced in Section 5.2.2.2. It is thus called OneOffEnergy.

The main steps (with modifications marked in red colour) are summed up in Algorithm 15. The only modification made is to check the remaining battery capacity (i) before searching for a new schedule (Line 16), or (ii) before committing a task copy (Line 4). Then, the algorithm changes a mode (if necessary) and schedules and/or commits tasks according to the current energy level in the battery and task priority.

Since the results in Section 5.2.4 showed that it is not necessary to test several ordering policies, the algorithm makes use of only one policy. Based on our previous results, the chosen ordering policy is the "Earliest Deadline".

5.3.3 Energy and Power Formulae

This section covers several formulae related to energy and power. These formulae require to know the number of executed tasks, which will be available after simulations, and will be used to assess the energy balance aboard CubeSats. We start with the formulae associated with the energy consumption and then continue with the ones related to the energy harvesting and storage. From the viewpoint of energy harvesting, a CubeSat experiences two periods: the daylight and the eclipse.

The energy consumption of a P-processor system when executing tasks during one hyperperiod (HT) and consuming $P_{executing}$ is as follows:

$$E_{HT_{executing}} = P_{executing} \cdot \sum_{i}^{\text{Scheduled tasks during } HT} et_i \tag{5.5}$$

Algorithm 15 Online energy-aware algorithm scheduling all tasks as aperiodic tasks (OneOffEnergy)

Input: Mapping and scheduling of already scheduled tasks, (task t_i, fault)
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**
      Check the current battery capacity
      if the task copy is authorised to be executed within the operating mode **then**
        Commit this task copy
      else
        Nothing to do
    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
    Check the current battery capacity
    if the task copy is authorised to be executed within the operating mode **then**
      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
    else
      Nothing to do

The energy consumption of the P-processor system when idle during one hyperperiod (HT) and consuming P_{idle} is as reads:

$$E_{HT_{idle}} = P_{idle} \cdot \left(P \cdot t_{HT} - \sum_{i}^{\text{Scheduled tasks during } HT} et_i \right) \tag{5.6}$$

where t_{HT} denotes the duration of one hyperperiod.

Summing Formulae 5.5 and 5.6, we get the energy consumption of the P-processor system during one hyperperiod (HT):

$$E_{HT} = E_{HT_{executing}} + E_{HT_{idle}} \tag{5.7}$$

and we can assess the power of the system consisting of P processors based on the energy consumption during one hyperperiod:

$$P_{system} = \frac{E_{HT}}{t_{HT}} \tag{5.8}$$

If we consider that the energy is consumed not only by processors aboard CubeSats but also by other components, such as a radio transmitter (TX) or a receiver (RX), the overall CubeSat power is as follows:

$$P_{CubeSat} = \sum_{i}^{\text{CubeSat components}} P_i \cdot (\text{duty cycle}) \tag{5.9}$$

The value of $P_{CubeSat}$ depends on the operating mode chosen by the algorithm.

As regards the harvested power, we consider that a CubeSat has a solar panel delivering $P_{harvested}$. The power available to recharge the battery P_{charge} is dependent on the current power consumption $P_{CubeSat}$ [28]:

$$P_{charge} = P_{harvested} \cdot \eta_d - P_{CubeSat} \tag{5.10}$$

where η_d denotes the transmission efficiency from solar panel to load [35].

The energy supplied by the solar panel during the daylight to charge the battery $E_{supplied}$ is as follows [28]:

$$E_{supplied} = P_{charge} \cdot (\text{time spent in the daylight within one orbit}) \tag{5.11}$$

This energy can be compared to the energy needed during the eclipse to power the CubeSat [28]:

$$E_{needed} = \frac{P_{charge} \cdot (\text{time spent in the eclipse within one orbit})}{\eta_e} \tag{5.12}$$

where η_e stands for the transmission efficiency from solar panel to battery and from battery to load [35].

To compute the energy stored in the battery, we make use of the following formula [28]:

$$E_{battery} = \eta_{battery} \cdot V_{battery} \cdot C_{Ah} \cdot DOD \tag{5.13}$$

where

— $\eta_{battery}$: battery transmission efficiency
— DOD: depth of discharge [7]
— $V_{battery}$: battery voltage
— C_{Ah}: battery capacity in Ah (battery capacity in Wh is computed as $C_{Wh} = V_{battery} \cdot C_{Ah}$)

7. *Depth of Discharge* (DOD) is the percentage of the capacity that has been removed from the fully charged battery [135].

5.3.4 Experimental Framework for CubeSats

This section is on our simulation scenario for CubeSats and defines metrics to evaluate the proposed algorithm.

5.3.4.1 Simulation Scenario

The data used in this experimental framework are based on Scenario APSS and they are therefore the same as presented in Section 5.2.3. Nevertheless, since every task has a new attribute, task priority tp_i (taking on three possible values: *high* (H), *middle* (M) and *low* (L)), the updated data are summarised in Table 5.9.

The data related to duration of various events are encapsulated in Table 5.10. We remind the reader that we distinguish two phases: the communication one and the no-communication one; and two periods: the eclipse and the daylight. Since the communication phase occurs at most once per CubeSat orbit and lasts for 10 minutes, which is shorter than the time spent in the daylight or eclipse, it is completed with the no-communication phase for the remaining time. Therefore, we distinguish two cases: (i) if there is no communication at all during a given period, the system functions in *the no-communication phase* only, and (ii) if a communication takes place during a given period, the system experiences both *the communication phase and the no-communication phase*. The former case is denoted by NCP, and the latter one by CP+NCP. We are aware these two cases are special cases because a period can change during the communication phase. Nevertheless, since Case (ii) represents the worst case from the energy point of view, the results would be better than our assessed values.

In our experiments, simulations start with the eclipse period. We consider two scenarios from the viewpoint when a communication takes place: one when it starts at 900 000 ms, i.e. during the eclipse, and another when it starts at 3 000 000 ms, i.e. during the daylight.

Table 5.9 – Set of tasks for Scenario APSS taking into account energy constraints

`Periodic tasks`

Function	**Task type** tt_i	**Phase** ϕ_i	**Period** T_i	**Execution time** et_i	**# tasks having** $tp_i = H$	**# tasks having** $tp_i = M$	**# tasks having** $tp_i = L$
Communication	D	U(0; T)	500 ms	U(1 ms; 10 ms)	1	0	1
Reading data	S	U(0; T)	1 000 ms	U(100 ms; 500 ms)	3	2	5
Telemetry	D	U(0; T)	5 000 ms	U(1 ms; 10 ms)	1	0	1
Storing data	S	U(0; T)	10 000 ms	U(100 ms; 500 ms)	2	2	3
Readings	D	U(0; T)	60 000 ms	U(1 ms; 10 ms)	1	0	1

`Sporadic tasks related to communication`

Function	**Task type** tt_i	**Phase** ϕ_i	**Period** T_i	**Execution time** et_i	**# tasks having** $tp_i = H$	**# tasks having** $tp_i = M$	**# tasks having** $tp_i = L$
Communication	S	U(0; T)	500 ms	U(1 ms; 10 ms)	5	0	41

`Aperiodic tasks`

Function	**Task type** tt_i	**Arrival time** a_i	**Execution time** et_i	**# tasks having** $tp_i = H$	**# tasks having** $tp_i = M$	**# tasks having** $tp_i = L$
Interrupts	D	U(0; 100 000 ms)	U(1 ms; 10 ms)	1	0	0

Finally, the energy and power data are summed up in Tables 5.11 and 5.12. While the former table contains data related to battery, processor and solar panels, the latter one presents information on other CubeSat components taken into account including their duty cycles in different modes. In order to obtain reasonable results, the components for 1U or 2U CubeSats were considered.

Table 5.10 – Simulation parameters related to time

Parameter	Value(s)	Note
Simulation duration	6 000 000 ms	100 hyperperiods
Orbit duration	5 700 000 ms	-
Period of daylight/eclipse	$\frac{2}{3}\cdot$ (orbit duration) / $\frac{1}{3}\cdot$ (orbit duration)	-
Duration of communication phase	600 000 ms	10 hyperperiods

Table 5.11 – Simulation parameters related to power and energy

Parameter	Value	Reference or note
Processor power consumption in Run mode when a processor is executing a task $P_{run(72MHz)}$	173 mW	[137]
Processor power consumption in Run mode when a processor is not executing a task $P_{run(125kHz)}$	4.6 mW	[137]
Processor power consumption in Standby mode $P_{standby}$	0.0083 mW	[137]
Harvested power $P_{harvested}$	2 300 mW	https://www.isispace.nl/product/isis-cubesat-solar-panels/
Transmission efficiency from solar panel to load η_d	0.8	[28, 35]
Transmission efficiency from solar panel to battery and from battery to load η_e	0.6	[28, 35]
Battery transmission efficiency $\eta_{battery}$	0.9	[28]
Depth of discharge DOD	0.2	[28]
Battery voltage $V_{battery}$	3.6 V	[65]
Battery capacity C	2 600 $mAh = 9.36\ Wh$	[65]
Maximum energy stored in the battery $E_{battery}$	6 065.28 J	Formula 5.13
Energy initially stored in the battery $E_{battery_{init}}$	$\{\frac{E_{battery}}{3}; \frac{2 \cdot E_{battery}}{3}; E_{battery}\}$	-

Table 5.12 – Other power consumption aboard a CubeSat taken into account (values of power from [112])

Component	Power	Duty cycle		
		Normal mode	Safe mode	Critical mode
RX	180 mW	100%	100%	100%
TX (communication phase)	2 800 mW	75%	75%	20%
TX (no-communication phase)	2 800 mW	0%	0%	0%
EPS	120 mW	100%	100%	100%

To model dynamic aspect, although task sets are defined for simulations in advance, they are unknown to algorithms until discrete simulation time has equalled arrival time.

To assess the number of tasks executed during one hyperperiod ($\sum_i^{\text{Scheduled tasks during } HT} et_i$) used in Formulae 5.5 and 5.6, 20 simulations of two hyperperiods were carried out. There were no changes of system modes, i.e. simulations were respectively conducted solely in normal, safe and critical modes.

To evaluate the algorithm taking into account the energy constraints, 20 simulations of 100 hyperperiods were realised. In this case, the algorithm changes modes according to the battery capacity, as mentioned in Table 5.6.

In both simulation cases, the obtained results were averaged.

Fault Generation

In this experimental framework, no simulation with fault injection was carried out due to time constraints because one simulation can take up to 9 hours. The fault tolerance of OneOffEnergy will be thereby evaluated in Section 5.3.7.5 for another application having energy constraints.

5.3.4.2 Metrics

To analyse the algorithm performances, we put into practice, similarly to the previous evaluations, the *rejection rate*, which is the ratio of rejected tasks to all arriving tasks, and the *system load* standing for the number of processors executing a task at a given time instant. The *processor load* is then computed as the system load divided by the number of processors. Since our system model considers that a processor can be in Run or Standby mode, we distinguish two processor loads: the one where all system processors are considered and the one where only processors in Run mode are taken into account. This differentiation allows us to better assess the utilisation of processors.

5.3.5 Results for CubeSats

In this section, we first calculate the energy balance on board of a CubeSat and then we assess the performances of OneOffEnergy.

5.3.5.1 Energy Balance

First of all, we compute the theoretical processor load for each mode (normal, safe and critical) when considering both maximum and mean execution times of each task based on Table 5.9. The results are depicted in Figures 5.24 representing such a processor load respectively for both communication phases as a function of the number of processors. Since several processors are switched in safe and critical modes, whose number is in Table 5.8, we compute the theoretical processor load from two points of view. On the one hand, we consider all system processors regardless of their operating mode and plot the results by solid lines. On the other hand, we take into account only processors operating in Run mode and represent the corresponding results by dashed lines.

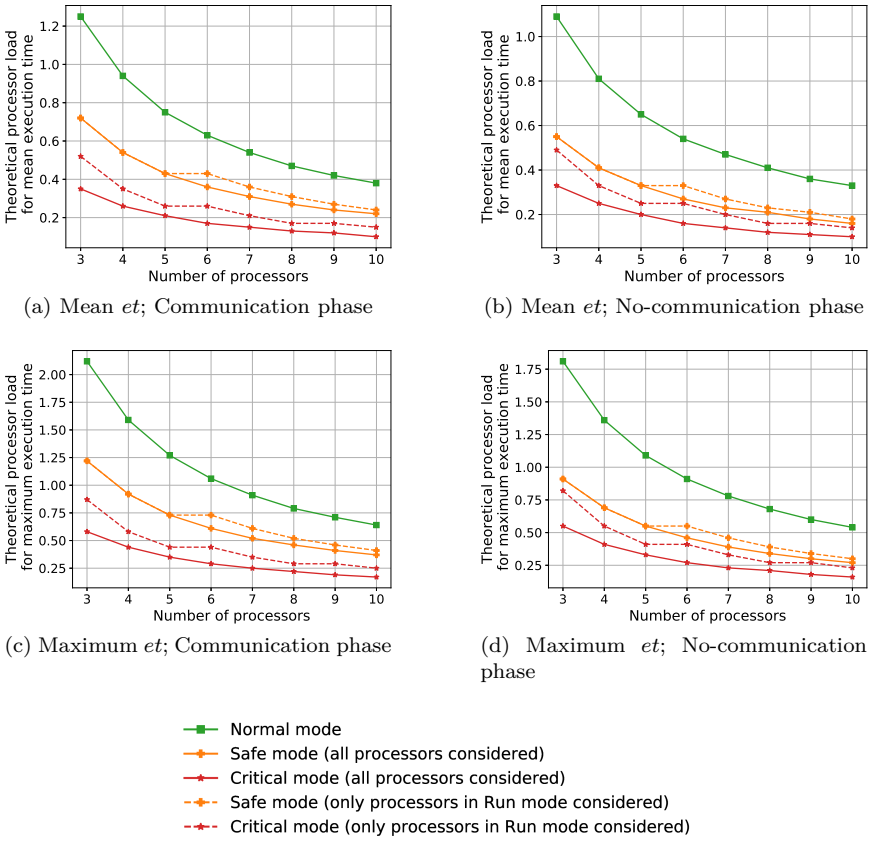
Figures 5.24 shows that the theoretical processor load depends on the mode and consequently on the number of tasks. In fact, the modes are chosen according to the battery capacity and each mode authorises only tasks having a given priority level or higher. It can be seen that, when considering only processors in Run mode, the theoretical processor load is sometimes constant (for example between 5 and 6 processors), which is due to the same task input regardless of the number of system processors and the floor function when switching processors to Standby mode.

Figures 5.25 depict the rejection rate for three modes (normal, safe and critical) as a function of the number of processors. In this figure and in this figure only, the rejection rate is computed considering only the authorised tasks, i.e. although low-priority tasks are not executed in a given mode, they do not exceptionally contribute to this metric in order to evaluate the ability to schedule the authorised tasks. Normal mode has the highest rejection rate because the system deals with all tasks no matter their priority. In safe and critical modes, only task with higher priority are authorised to be executed. Consequently, there are less tasks to be scheduled and lower or none rejection rate.

Next, based on the data of the executed tasks, we compute the energy consumption using Formulae 5.5 and 5.6. To represent the worst-case scenario, we consider that all processors are always in Run mode and none of them is put into Standby mode. Figures 5.26 show the useful and idle energy consumption of CubeSat processors in three system modes during two hyperperiods as a function of the number of processors during the communication phase. It is observed that once processors can accommodate all tasks, the useful energy consumption remains the same and the higher the number of processors, the higher the idle energy consumption. The energy consumption due to idle processors is negligible when compared to the one when processors are executing.

The simulations during the no-communication phase were also carried out. The results are qualitatively similar to the ones obtained during the communication phase and the values are slightly lower.

Once the energy consumption of processors is evaluated, we determine the corresponding power of processors using Formula 5.8. Taking into account also all CubeSat components, mentioned in Table 5.12,



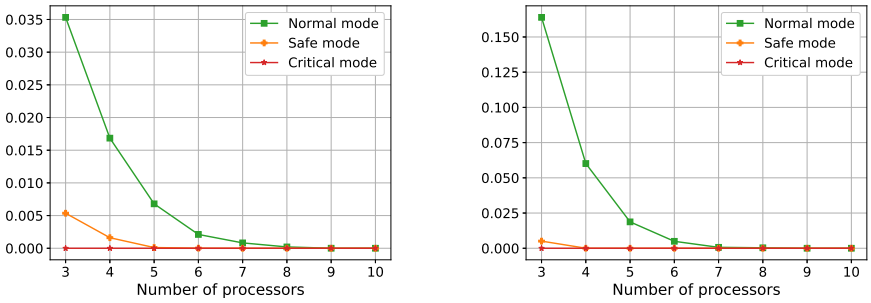
(a) Mean *et*; Communication phase

(b) Mean *et*; No-communication phase

(c) Maximum *et*; Communication phase

(d) Maximum *et*; No-communication phase

Figure 5.24 – Theoretical processor load when considering maximum and mean execution times (*et*) of each task



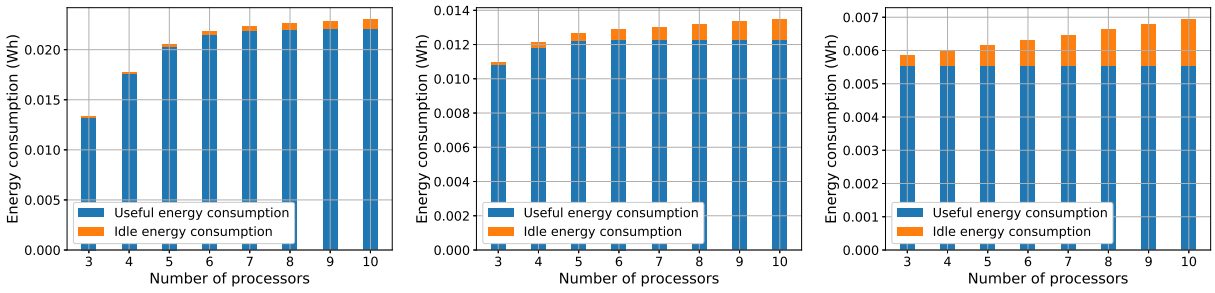
(a) Communication phase

(b) No-communication phase

Figure 5.25 – Rejection rate for three system modes as a function of the number of processors

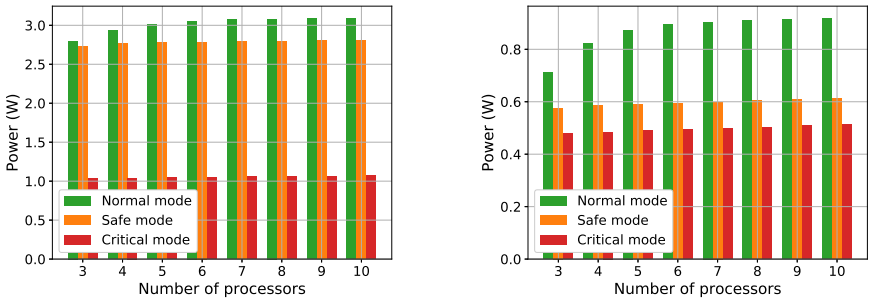
Formula 5.9 gives the overall CubeSat power consumption. The results are depicted in Figures 5.27 representing the power in three system modes as a function of the number of processors.

We notice that the power is significantly higher during the communication phase than during the no-communication phase. This difference is due to the radio communication transmitter consuming 2.8 W and its duty cycle of 75% in normal and safe modes during the communication phase. Moreover, the power required in safe and critical modes is lower than the one in normal mode.



(a) Normal mode (b) Safe mode (c) Critical mode

Figure 5.26 – Useful and idle energy consumptions during two hyperperiods as a function of the number of processors (communication phase)



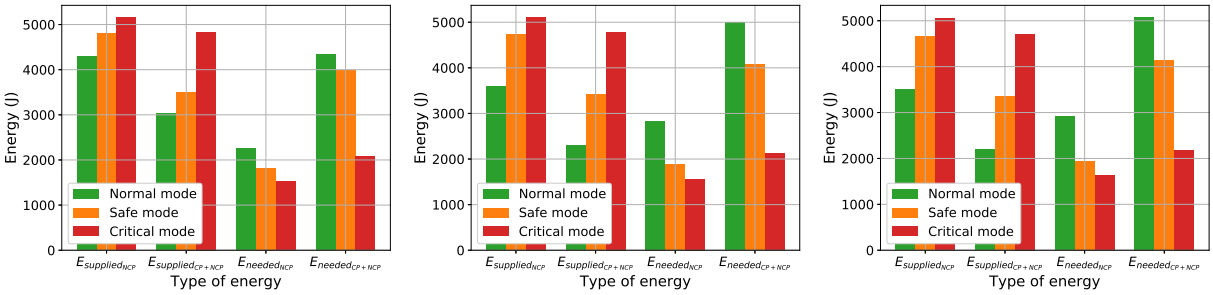
(a) Communication phase (b) No-communication phase

Figure 5.27 – CubeSat power consumption in three system modes as a function of the number of processors

Finally, we evaluate the energy balance aboard a CubeSat. Thus, we compare the energy supplied by the solar panel during the daylight to charge the battery $E_{supplied}$ as expressed by Formula 5.11 and the energy needed during the eclipse to power the CubeSat E_{needed}, as given by Formula 5.12. Since the communication phase occurs at most once per CubeSat orbit and lasts only for 10 minutes, which corresponds to 10 hyperperiods, we examine two cases (as introduced in Section 5.3.4.1): (i) if there is no communication at all during a given period, the system functions in *the no-communication phase* only, and (ii) if a communication takes place during a given period, the system experiences both *the communication phase and the no-communication phase*. The former case is denoted by NCP, while the latter one by CP+NCP. The results are plotted in Figures 5.28 respectively depicting two cases for $E_{supplied}$ and E_{needed} for a system composed of 3, 6 or 9 processors.

The higher the number of processors, the higher the power $P_{CubeSat}$, the higher the energy needed in the eclipse E_{needed}, the lower the power to recharge the battery P_{charge} and therefore the lower the value of the energy $E_{supplied}$. It is worth noticing the battery capacity ($E_{battery} = 6.1\,kJ$) is sufficient to provide enough energy for $E_{needed} < E_{battery}$ regardless of system mode. On the one hand, while normal mode functions well during the no-communication phase because the energy supplied $E_{supplied}$ covers all energy expenses E_{needed}, normal mode experiences energy shortages during the communication phase owing to $E_{needed_{CP+NCP}} > E_{supplied_{NCP}}$ all the time for any number of processors. On the other hand, although the communication phase is very demanding, the energy supplied in critical mode $E_{supplied}$ is always sufficient to the demand in such a mode and there is never lack of energy even in the worst-case scenario (CP+NCP, $P = 10$).

We conclude that for CubeSats, whose payload does not require a lot of power, e.g. measurement of electron density, it is useless to develop a specific algorithm optimising the schedule taking into account energy harvesting and consumption. In fact, a simple check of the energy level available in the battery and several operating system modes are sufficient.



(a) $P = 3$ (b) $P = 6$ (c) $P = 10$

Figure 5.28 – Comparison of the energy supplied by the solar panel during the daylight to charge the battery and the energy needed during the eclipse to power the CubeSat

5.3.5.2 Algorithm Performances

The preceding section analysed results based on the evaluation of the energy balance in each system mode. The aim of this section is to evaluate the performances of OneOffEnergy when operating on board of a CubeSat.

We analyse two scenarios. The first one considers that the communication phase occurs during the eclipse, while the second one considers it during the daylight. In both cases, the initial battery capacity is set at one third of the maximum battery capacity. We conduct simulations for 100 hyperperiods, which is more than one orbit duration.

Figures 5.29 and 5.31 plot the energy level in the battery against time. At the bottom of the figures, black and yellow colours respectively indicate the eclipse and the daylight. The vertical dashed lines mark a change of mode.

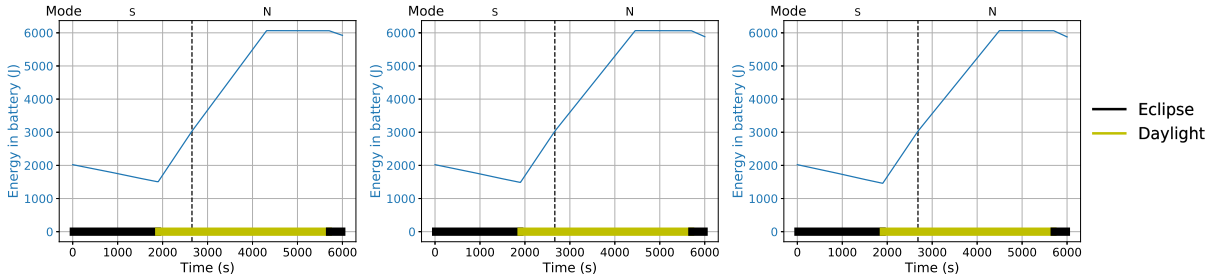
Regardless of when the communication phase occurs, Figures 5.29 and 5.31 show that no energy shortage occurs and a CubeSat can operate in one of its modes. Furthermore, the battery is charged from approximately one sixth to its full capacity within less than one daylight, which confirms that the battery can be sufficiently replenished.

Figures 5.30 and 5.32 represent the system and processor loads in the course of time. As defined in Section 5.3.4.2, the figures show the processor load when all system processors are considered (blue curve) and the one taking into account processors in Run mode only (red curve). In order to enhance the readability, the loads are computed within the window of size 10 s within one mode and averaged. The eclipse and daylight periods and the mode changes are also plotted.

The system and processor loads remain almost constant in a given phase, which is due to the execution of the same tasks related to CubeSat housekeeping. We notice that when the communication phase occurs (at $900000ms$ if it takes place during the eclipse, or at $3000000ms$ otherwise) and lasts for 10 hyperperiods, i.e. $600000ms$, the system and processor loads are higher to satisfy the demand and schedule tasks related to the communication.

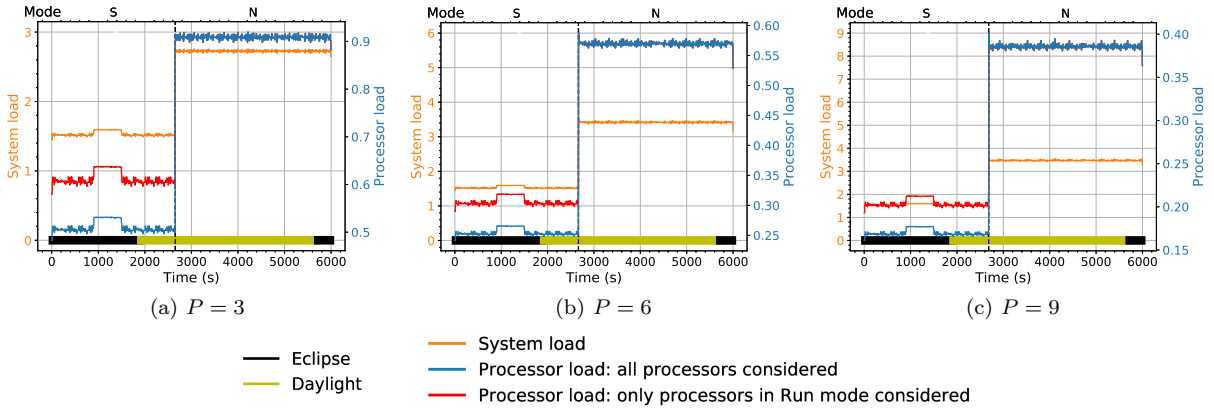
As stated in Table 5.11, simulations (regardless of when the communication phase occurs) with the initial battery capacity respectively equal to $\frac{2}{3} \cdot E_{battery}$ and $E_{battery}$ were also carried out. Thanks to higher initial battery capacity, the system spends more time in normal mode instead of safe one and it consequently executes more tasks and its rejection rate is lower. Actually, due to energy savings, tasks having the lowest priority are automatically rejected in safe mode, which increases the rejection rate.

The dependency of the rejection rate on the number of processors and the initial energy level in the battery when the communication phase occurs respectively in the eclipse and in the daylight are respectively depicted in Figures 5.33. It can be observed that if the communication phase takes place during the eclipse, the rejection rate is higher (up to almost 50% for a 3-processor system with $E_{battery_{init}} = \frac{1}{3} \cdot E_{battery}$) than if it occurs during the daylight (17% for a 3-processor system with $E_{battery_{init}} = \frac{1}{3} \cdot E_{battery}$) due to higher energy consumption.



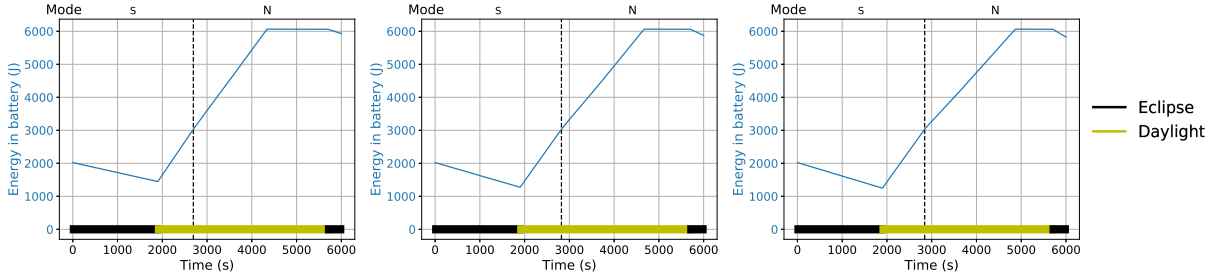
(a) $P = 3$ (b) $P = 6$ (c) $P = 9$

Figure 5.29 – Energy in the battery against time (communication phase in the eclipse; $E_{battery_{init}} = \frac{1}{3} \cdot E_{battery}$) [8]



(a) $P = 3$ (b) $P = 6$ (c) $P = 9$

Figure 5.30 – System and processor loads against time (communication phase in the eclipse; $E_{battery_{init}} = \frac{1}{3} \cdot E_{battery}$)



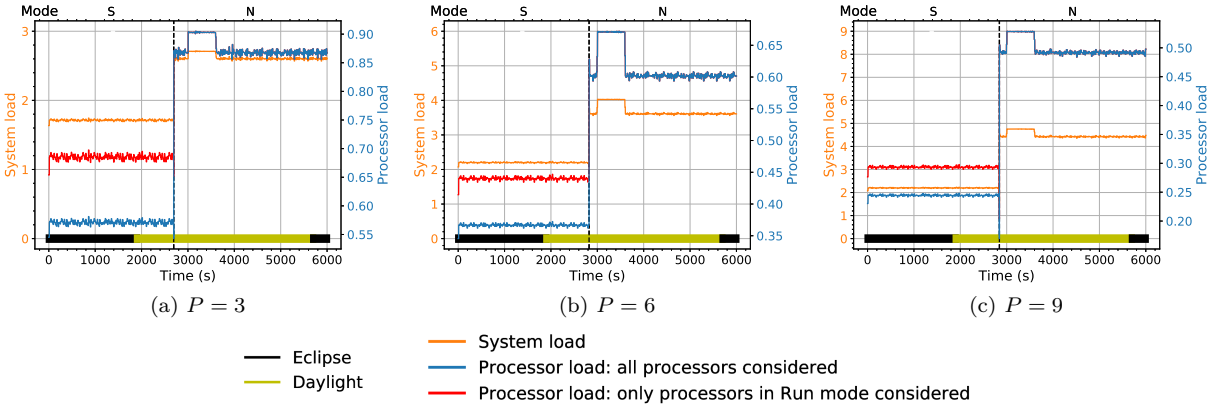
(a) $P = 3$ (b) $P = 6$ (c) $P = 9$

Figure 5.31 – Energy in the battery against time (communication in the daylight; $E_{battery_{init}} = \frac{1}{3} \cdot E_{battery}$)

In compliance with the results from the preceding section, we conclude that CubeSats containing only low power consumption payload do not take a risk to experience any energy shortage because the supplied energy covers all energy expenses. Their energy balance is overestimated in order not to jeopardise the mission because of an energy issue.

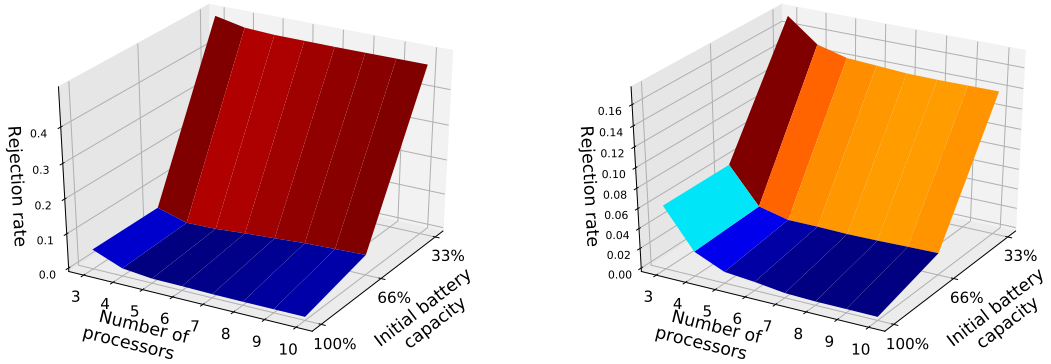
To further analyse the performances of the proposed algorithm, we study another experiment scenario. Our aim is to compare OneOffEnergy with other simpler algorithms and evaluate its efficiency and fault tolerance.

8. S and N denote safe mode and normal mode, respectively.



(a) $P = 3$ (b) $P = 6$ (c) $P = 9$

Figure 5.32 – System and processor loads against time (communication phase in the daylight; $E_{battery_{init}} = \frac{1}{3} \cdot E_{battery}$)



(a) Communication phase in the eclipse (b) Communication phase in the daylight

Figure 5.33 – Rejection rate as a function of the number of processors and the initial battery energy

This study is not easily feasible using CubeSat Scenario APSS for two main reasons. First, the results show that CubeSats, whose payload does not require a lot of power, do not encounter an energy shortage. Consequently, it would be difficult to evaluate the subtle difference between OneOffEnergy and another algorithm that never puts processors in Standby mode. Second, simulations based on the CubeSat scenario require a lot of time to be carried out and one simulation can last up to for 9 hours. Since we consider only 100 hyperperiods per simulation, which is slightly more than one orbital period (95 hyperperiods), it is not possible to reduce the number of hyperperiods because at least one orbital period is necessary to assess the energy balance.

5.3.6 Experimental Framework for Another Application

Taking into account the remarks made at the end of the preceding section, we consider another experiment scenario, which is presented in this section, along with employed metrics.

5.3.6.1 Simulation Scenario

While the simulation scenario introduced in Section 5.3.4 is based on real CubeSat data, the data for the simulation scenario presented in this section were synthetically generated, except the energy and power

data corresponding to the STM32F103 processor. The values were chosen in order to have reasonable simulation times. For example, the periods of the daylight and eclipse were shortened. Moreover, the battery capacity is not overestimated, as it is the case aboard CubeSats to avoid energy shortages.

The current scenario is based on parameters that are encapsulated in Tables 5.13, 5.14 and 5.15. The first table gathers general parameters, the second one concentrates parameters related to time and the third one sums up the energy and power data.

Similarly to the CubeSats scenarios, the task input is always the same regardless of the number of processors and the simulations start in the eclipse.

Table 5.13 – Simulation parameters

Parameter	**Value(s)**
Number of simple tasks	15 000
Number of double tasks	15 000
Number of processors P	3 – 10
Task priority tp	Uniform(Low, Middle, High)

Table 5.14 – Simulation parameters related to time

Parameter	**Distribution**	**Value(s) in** ms
Simulation duration	-	1 500 000
Period of daylight/eclipse	-	400 000 / 400 000
Arrival time a	Exponential	$\frac{\text{simulation duration}}{\text{number of simple tasks+number of double tasks}}$
Execution time et	Uniform	10 – 100
Deadline d	Uniform	$[\![a + 3 \cdot et; a + 10 \cdot et]\!]$

Table 5.15 – Simulation parameters related to power and energy (inspired by values from Table 5.11)

Parameter	**Value**
Processor power consumption in Run mode when a processor is executing a task $P_{run(72MHz)}$	173 mW
Processor power consumption in Run mode when a processor is not executing a task $P_{run(125kHz)}$	4.6 mW
Processor power consumption in Standby mode $P_{standby}$	0.0083 mW
Harvested power $P_{harvested}$	750 mW
Maximum energy stored in the battery $E_{battery}$	100 J
Energy initially stored in the battery $E_{battery_{init}}$	90 J

To model dynamic aspect, although task sets are defined for simulations in advance, they are unknown to algorithms until discrete simulation time has equalled arrival time.

To evaluate the OneOffEnergy performances, 20 simulations were conducted and the obtained values were averaged.

Fault Generation

Similarly to Section 5.2.3.1, 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. We remind the reader that the worst estimated fault rate in the real space environment is 10^{-5} fault/ms [118]. We thereby evaluate the algorithm performances 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.

5.3.6.2 Metrics

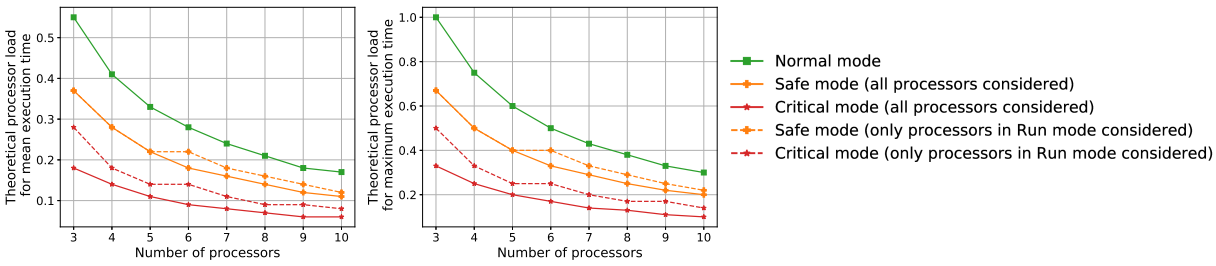
Similarly to Section 5.3.4.2, we make use of the *rejection rate*, *system throughput*, *system load* and *processor load*.

5.3.7 Results for Another Application

In this section, we first compute the theoretical processor load and then evaluate the performances of OneOffEnergy, which are compared to other algorithms to show the benefits of our devised algorithm. Finally, simulations with fault injection are conducted to estimate the algorithm behaviour in a harsh environment.

5.3.7.1 Theoretical Processor Load

We compute the theoretical processor load for each system mode when considering both maximum and mean task execution times. The results are plotted in Figures 5.34 depicting such processor loads as a function of the number of processors. Taking into account that several processors can be put into safe and critical modes (the exact number is mentioned in Table 5.8), the theoretical processor load is computed as follows: (i) all system processors are considered regardless of their operating mode, and (ii) only processors operating in Run mode are taken into account.



(a) Mean *et* (b) Maximum *et*

Figure 5.34 – Theoretical processor load when considering maximum and mean execution times (*et*) of each task

As we have already concluded for Scenario APSS in Section 5.3.5.1, the theoretical processor load depends on the mode and consequently on the number of tasks. We stress that the represented values do not consider any changes of modes but in reality the modes change, which affects the real processor load.

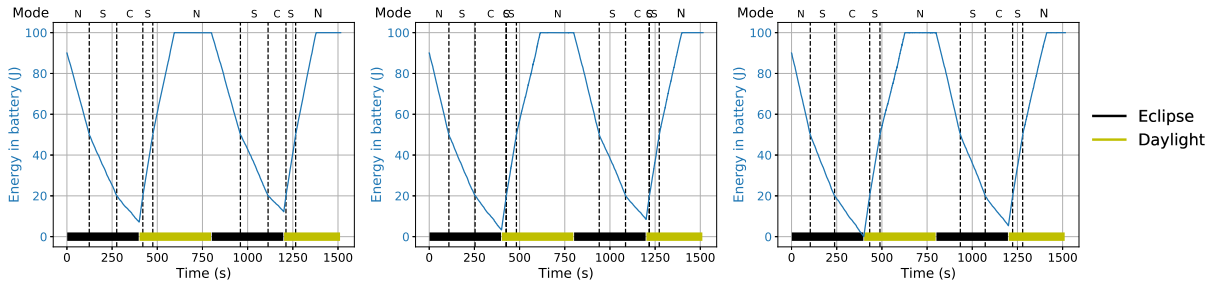
5.3.7.2 Analysis of OneOffEnergy

In this section, we evaluate the performances of OneOffEnergy.

Figures 5.35 plot the energy level in the battery against time and show when a mode changes by means of the vertical dashed lines. It may happen that, when transiting from lower mode to higher one (e.g. from critical (C) mode to safe (S) one), the system returns back to the previous mode because the current energy level in the battery is again temporarily under the threshold. That is seen for example in Figure 5.35b at 426 ms. To illustrate the eclipse and daylight periods, the bottom line in the figure respectively indicates black and yellow colours.

The tendency of the evolution of the energy in the battery depends on the mode, e.g. in normal mode the energy is consumed more and charged less than in other modes. The higher the number of processors, the more noticeable the difference and less time spent in normal mode due to higher energy consumption.

To better evaluate the time spent in system modes, we add up the times spent in each mode (normal, safe and critical) or in the state without energy, if applicable. The results are plotted in Figure 5.38a,



(a) $P = 3$ (b) $P = 6$ (c) $P = 9$

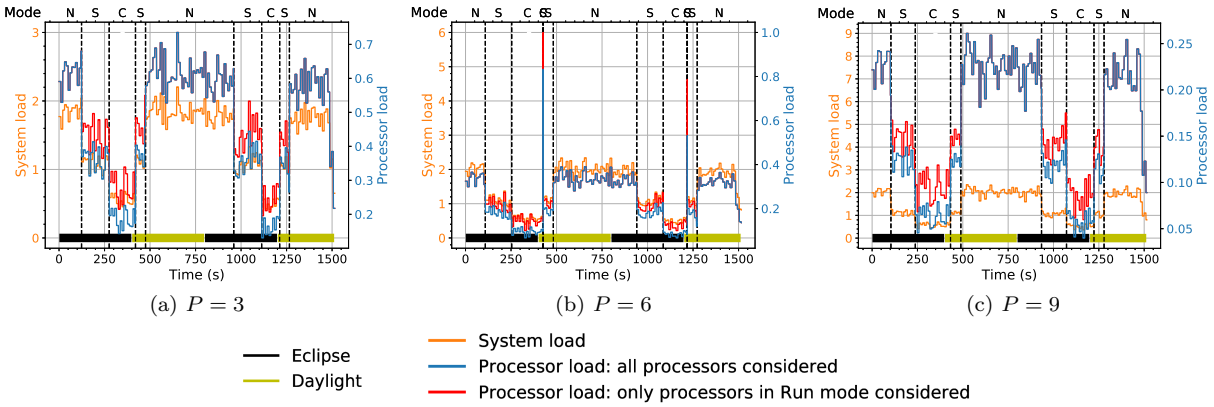
Figure 5.35 – Energy in the battery against time [9]

which represents such times within one simulation duration as a function of the number of processors. As we have already concluded previously, when the number of processors increases, the times spent in normal and then safe mode are shorter.

As a consequence of spending more time in normal mode instead of safe one or critical one, the system executes more tasks and its rejection rate is lower. Actually, due to energy savings in safe and critical modes, low-priority tasks are automatically rejected in safe and critical modes, which increases the rejection rate. Figure 5.40a representing the rejection rate as a function of the number of processors shows that the lower the number of processors, the lower the rejection rate because the system consumes less energy.

Figures 5.36 depict the system and processor loads in the course of time. As defined in Section 5.3.6.2, we distinguish the processor load when all system processors are considered (blue curve) and the one taking into account processors in Run mode only (red curve). To enhance the readability, the loads are computed within the window of size 10 s within one mode and averaged. The eclipse and daylight periods are indicated by the black-and-yellow line. The mode changes are plotted by the vertical dashed lines.

In general, the higher the number of processors, the lower the processor load because the number of tasks to be executed is unchanged and the system load thereby remains the same. When several processors are switched into Standby mode (during safe or critical mode), the load of processors being in Run mode is higher than the one considering all system processors. Moreover, the system and processor loads do not significantly vary within one mode because the number of tasks to be executed does not change.



(a) $P = 3$ (b) $P = 6$ (c) $P = 9$

Figure 5.36 – System and processor loads against time

9. N, S and C denote normal, safe and critical modes, respectively.

5.3.7.3 Assessment of Standby Mode

To assess the benefit of putting processors in Standby mode, we compare OneOffEnergy to the algorithm being the same as OneOffEnergy except that the processors are never switched to Standby mode. It means that they are always in Run mode and operate at the maximum frequency ($72\,MHz$), if executing a task, or at the minimum frequency ($125\,kHz$) otherwise.

Since the processors of such an algorithm are never put into Standby mode, the power consumption is higher, which shortens the time spent in normal mode, as shown in Figure 5.38b representing the sum of times spent in different system modes or in the state without energy. While OneOffEnergy never experiences the energy shortage up to $P = 9$ (as presented in Figure 5.38a), another algorithm encounters it when a system has more than 8 processors due to its higher energy consumption.

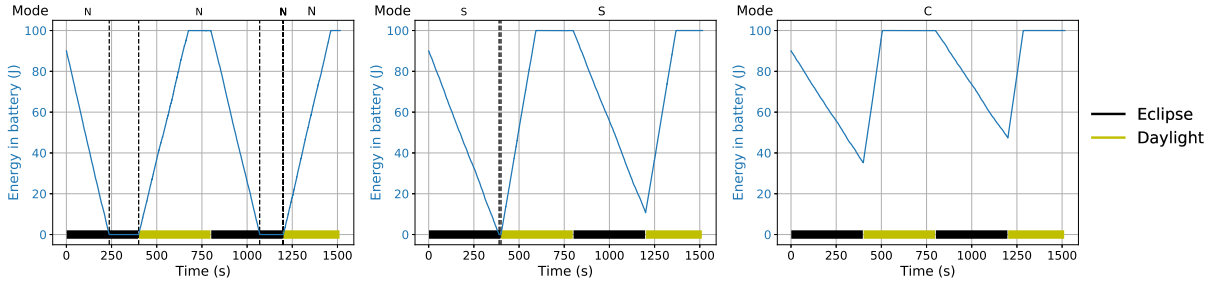
Figure 5.40a depicts the rejection rate as a function of the number of processors. The higher the number of processors, the larger the gap between the rejection rate of OneOffEnergy and the other algorithm. When the system makes use of Standby mode, it functions longer in normal mode instead of safe one or critical one and it consequently executes more tasks and its rejection rate is lower. Recall that, due to energy savings, low-priority tasks are automatically rejected in safe and critical modes, which increases the rejection rate.

The energy savings thanks to Standby mode are not negligible and can avoid a lack of energy. Its use is therefore well appropriate for systems with energy constraints.

5.3.7.4 Assessment of System Operation

In this section, we evaluate the system operation in the course of time. We compare OneOffEnergy to algorithms operating in only one mode (normal, safe or critical). Such algorithms never put processors into Standby mode, which means that the processors are all the time in Run mode operating at the maximum frequency ($72\,MHz$), if executing a task, or at the minimum frequency ($125\,kHz$) otherwise.

First, we observe the energy level in the battery against time and potential energy shortages in Figures 5.37 for a 9-processor system. This system was chosen in order to have higher energy consumption than a system with only a few processors. While the energy level in the battery is never depleted in critical mode, it is the case in safe mode (at $389\,s$) and in normal mode (regularly at the end of eclipse).



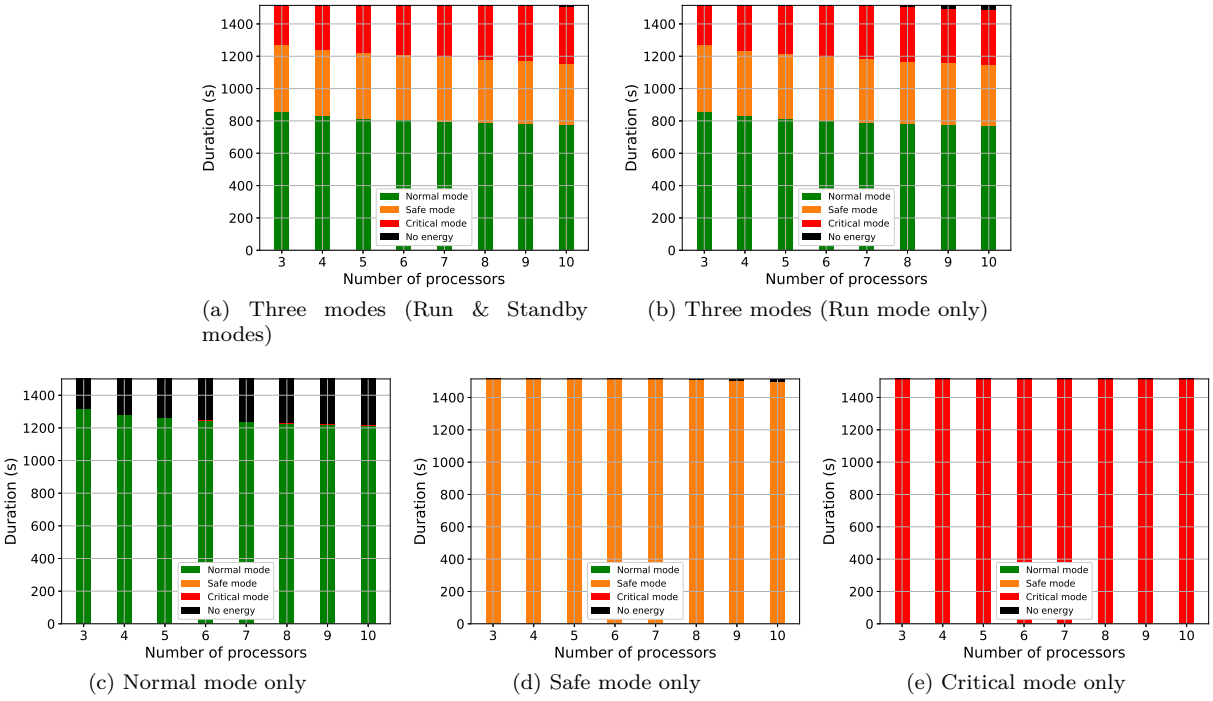
(a) Normal mode only (b) Safe mode only (c) Critical mode only

Figure 5.37 – Energy in the battery against time ($P = 9$)

Figures 5.38c, 5.38d and 5.38e depict the time spent in the respective system mode or alternatively the energy shortage. The operation mainly depends on the energy consumption. When the system consists of more processors, its operation time is shorter for normal and safe modes, or unchanged for critical mode.

Next, Figures 5.39 depict the system and processor loads in the course of time. Since none processor is switched into Standby mode, the processor load always considers all system processors. To improve the readability, the loads are computed within the window of size $10\,s$ within one mode and averaged.

The system and processors loads decreases when the system operates in stricter mode, e.g. while all tasks are authorised to be executed in normal mode, the ones with the lowest priority are forbidden in safe mode. Note that curves for system and processor loads in Figure 5.39a overlap each other.



(a) Three modes (Run & Standby modes)

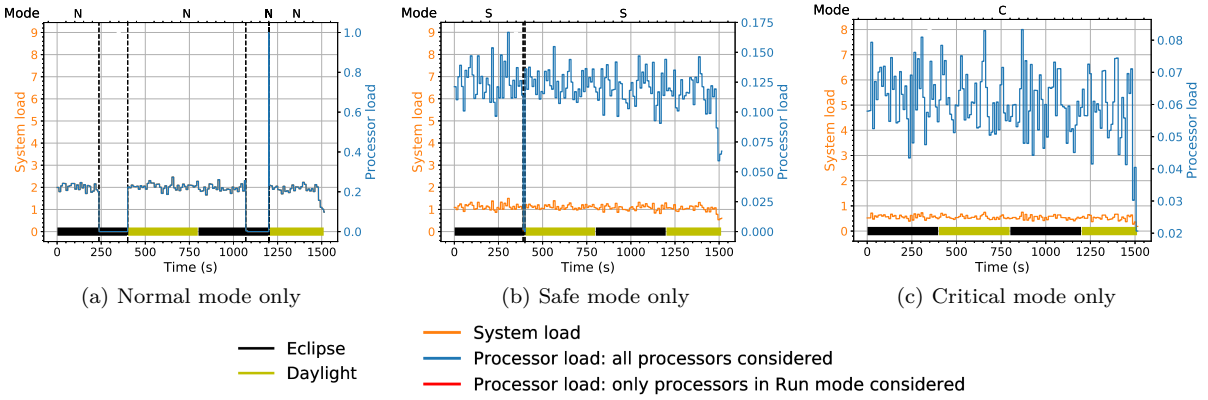
(b) Three modes (Run mode only)

(c) Normal mode only

(d) Safe mode only

(e) Critical mode only

Figure 5.38 – Overall time spent in different system modes



(a) Normal mode only

(b) Safe mode only

(c) Critical mode only

Figure 5.39 – System and processor loads against time ($P = 9$)

The rejection rates and processor loads (considering all system processors) as a function of the number of processors for OneOffEnergy and other aforementioned comparing algorithms are depicted in Figures 5.40.

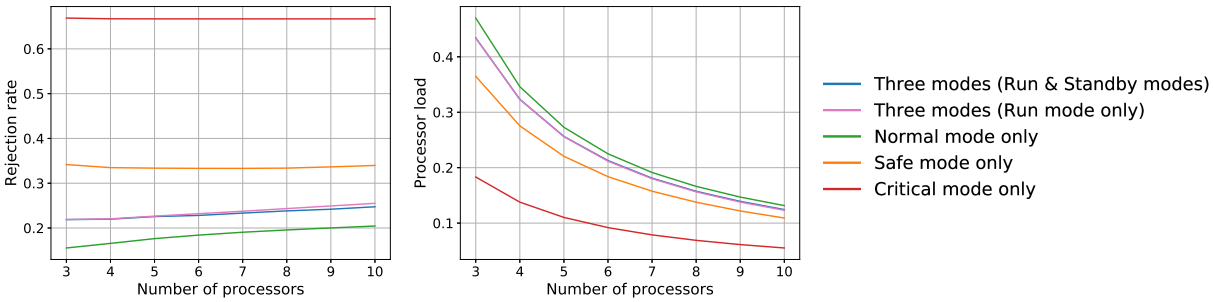
The rejection rate mainly depends on the mode because only tasks having a given level of priority or higher can be executed, e.g. in critical mode only tasks with the highest priority are authorised. In our experimental framework presented in Table 5.13, three task priorities (high, middle and low) are uniformly distributed. Consequently, approximately one third of tasks in safe mode and two thirds of tasks in critical mode are automatically rejected due to the task priority restrictions. While all tasks with the highest priority are authorised in critical mode and they are all scheduled, in safe mode there are several tasks, which are authorised to be executed but finally not scheduled due to not enough resources

(when the number of processors is low) or a lack of energy (when the number of processors is higher), which explains the slight variation of the rejection rate as a function of the number of processors.

Regarding the remaining curves, the higher the number of processors, the higher the rejection rate because the energy consumption increases with the number of processors, which forces the system to operate more frequently in safe and critical modes and therefore automatically reject low-priority tasks.

One may conclude that normal mode is better than our proposed algorithm (OneOffEnergy), which puts into practice three system modes (normal, safe and critical) and takes advantage of Run and Standby processor modes, because its rejection rate is lower, e.g. by 19% for a 6-processor system. Nevertheless, it is necessary to realise that normal mode does not take into account task priorities and regularly experiences energy shortages. By contrast, OneOffEnergy authorises to execute tasks with a given priority level based on the current energy level in the battery to optimise the energy consumption, avoids lacks of energy and performs at least tasks with the highest priority. Therefore, our devised algorithm presents a reasonable trade-off between the system operation, such as the number of tasks and their priority, and the energy constraints.

As regards the processor load, when the system has more processors, it decreases because the task input is always the same. Its values are also related to the rejection rate because the lower the rejection rate, the higher the processor load, except for critical and safe modes only. When we compare the real processor load (Figure 5.40b) with the theoretical one (Figures 5.34), the processor loads of safe and critical modes are approximately equal owing to almost no task rejection. For normal mode, the theoretical processor load based on mean execution time is higher than the real one because the system is not operational all the time due to lacks of energy.



(a) Rejection rate

(b) Processor load (all processors considered)

Figure 5.40 – System metrics as a function of the number of processors

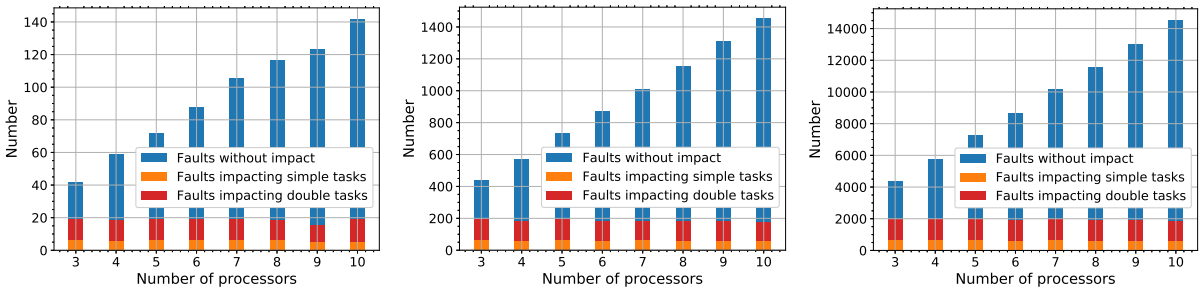
5.3.7.5 Simulations with Fault Injection

In this section, we assess the fault tolerance of OneOffEnergy.

Figures 5.41 depict the total number of faults against the number of processors, while the total number is the sum of the faults without impact, faults impacting simple tasks and faults impacting double tasks. The higher the fault rate per processor, the higher the number of impacted tasks. Most faults have no impact, the number of impacted tasks is rather low and remains almost constant because the data set at the input is always the same and does not require many resources. As there is the same number of simple and double tasks in our experimental framework (Table 5.13), the number of impacted simple tasks is theoretically one third and the one of double tasks is two thirds. The experimental results are in accordance with these theoretical values.

Figures 5.42 plot the rejection rate, system throughput and processor load as a function of the number of processors. When the number of processors increases, the rejection rate is higher and other two metrics decrease because the energy consumption is higher [10].

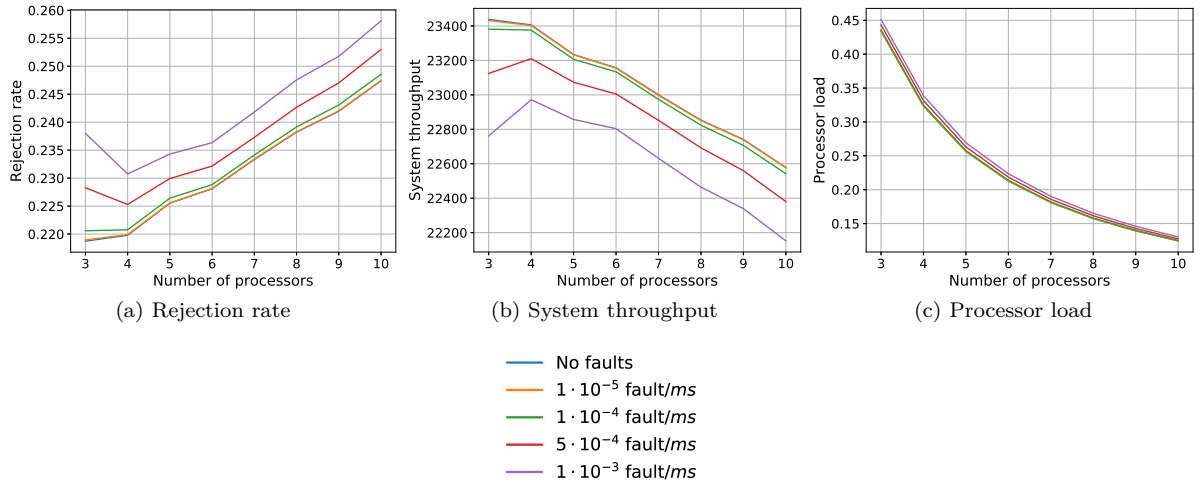
10. In Section 5.3.7.2, it was demonstrated that when the number of processors increases, the rejection rate is higher



(a) $1 \cdot 10^{-5}$ fault/ms (b) $1 \cdot 10^{-4}$ fault/ms (c) $1 \cdot 10^{-3}$ fault/ms

Figure 5.41 – Total number of faults (injected with a given fault rate) against the number of processors

The higher the fault rate, (i) the higher the rejection rate for there are less free slots due to backup copies, which are executed and not deallocated, especially for 3-processor systems, (ii) the lower the system throughput because less tasks are correctly executed, and (iii) the higher the processor load owing to execution of the backup copies. We remind the reader that the rejection rate characterises the schedulability, while the system throughput counts the number of correctly executed tasks. In order not to reduce the readability of Figure 5.42b, the maximum system throughput equal to 30 000 is not plotted. Furthermore, we note that the proposed algorithm performs well 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 [118]).



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

Figure 5.42 – System metrics at different fault injection rates as a function of the number of processors (OneOffEnergy)

5.3.8 Summary

This chapter presents and evaluates online scheduling algorithms for CubeSats with aim to make these small satellites fault tolerant. We propose to take advantage of multiprocessor architecture and gather all CubeSat processors on one board. This multiprocessor system makes use of one of our devised algorithms, which schedules all tasks on board, detects faults and takes appropriate measures in order to deliver correct results.

because the energy consumption is higher and the system spends less time in normal and safe modes. Consequently, low-priority tasks are automatically rejected due to energy savings in safe and critical modes.

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. Whereas these two algorithms do not take energy constraints into account, the last proposed algorithm called OneOffEnergy is energy-aware. All algorithms can use different ordering policies to sort a task queue.

The performances of OneOff and OneOff&Cyclic were studied for three different scenarios based on two real CubeSat scenarios. It was shown that they are influenced by the system load and proportions of simple and double tasks to all tasks to be executed. Overall, the "Earliest Deadline" and "Earliest Arrival Time" ordering policies perform well (measured by means of the rejection rate) for OneOff and the "Minimum Slack" ordering policy for OneOff&Cyclic. Furthermore, evaluating several ordering policies at every scheduling search (method called "All techniques") does not perform better than aforementioned policies and its main drawback is longer algorithm run-time induced by multiple scheduling searches before choosing the one minimising the rejection rate. Moreover, it is useless to consider systems with more than six processors because there is already no rejection rate for well chosen ordering policies and it is better not to oversize the system.

Although the number of scheduling attempts is significantly lower for OneOff&Cyclic than for OneOff, the former algorithm carries out a search for a new schedule more quickly than the latter one. The scheduling time is shorter during the no-communication phase than during the communication phase for there are less tasks to be scheduled. The method to reduce the number of scheduling searches cuts down this number but at the cost of higher rejection rate, which is not compatible with our objective function, to minimise the rejection rate, and the method is not used any longer.

The results demonstrate that OneOff&Cyclic does not generally perform as well as OneOff (in terms of the rejection rate and scheduling time) in the context of CubeSats but it can be put into practice in other applications with much more benefit (for example in embedded systems with real-time and energy constraints), where there are less scheduling triggers (less faults or less aperiodic tasks or less changes in set of periodic tasks) than in the studied application.

Therefore, we suggest that teams, which design their CubeSats gathering all processors on one board, should make use of OneOff when choosing a no-energy-aware algorithm.

The second part of this chapter is dedicated to an energy-aware version of OneOff. This modified algorithm called OneOffEnergy takes advantage of two processor operating modes (Run and Standby) and it considers three system modes (normal, safe and critical) depending on the energy available in the battery. The new enhanced algorithm was not only assessed for CubeSats but also in the context of another application having energy constraints.

The energy balance for CubeSat Scenario APSS showed that the communication phase requires a lot of energy mainly due to high power consumption of the transmitter ($2.8W$). Although the communication phase lasts for 10 minutes, which is rather short duration compared to the orbital period of 95 minutes and it occurs approximately six times out of fifteen daily orbits around the Earth, it may cause a lack of energy. Nevertheless, when OneOffEnergy is put into practice and a CubeSat operates within one of the following modes: normal, safe and critical, the energy shortage does not take place. In fact, the energy supplied is always sufficient to the demand in critical mode for values computed even in the worst-case scenario, i.e. when the communication phase occurs during the no-communication phase for a 10-processor system.

We state that CubeSats, whose payload does not require a lot of power, e.g. measurement of electron density, do not experience any energy shortage. Consequently, they can use a simple algorithm, such as OneOffEnergy, to check the energy level in the battery capacity and choose one of the system modes (normal, safe or critical) according to the current available energy level.

Since the CubeSat scenario does not allow us to assess all performances of our proposed energy-aware algorithm, we carry out simulations also for another energy-constrained application and compare the performances of OneOffEnergy with other simpler algorithms. The main differences are as follows: (i) the data for the application were synthetically generated (instead of real data for CubeSats), (ii) the time

spent in the daylight and eclipse were shortened to reduce the duration of the simulation, and (iii) the battery capacity is not overestimated, as it is the case aboard CubeSats to avoid energy shortages.

We found out that putting the processors in Standby mode brings energy savings. These savings allow the system to operate longer in normal or safe modes and consequently avoid automatic rejection of low-priority tasks. Although a system operating in normal mode only has lower rejection rate than the one using OneOffEnergy (for example by 19% for the 6-processor system), it is not able to run all the time due to limited energy resources. In contrast, OneOffEnergy chooses one of the system modes (normal, safe or critical) according to the available energy stored in the battery, executes tasks with appropriate priorities to optimise the energy consumption and avoids lacks of energy. Therefore, our algorithm presents a reasonable trade-off between the system operation, such as the number of tasks and their priority, and the energy constraints.

Finally, it was found that all three presented algorithms (OneOff, OneOff&Cyclic and OneOff-Energy) perform well also in a harsh environment.

The achievements of this chapter were published in Proceedings of the 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) and of the Euromicro Conference on Digital System Design (DSD), both held in 2020.

Chapter 6

Conclusions

The thesis is aimed at providing multiprocessor systems with fault tolerance and is in particular concerned with online mapping and scheduling of tasks on such systems in order to improve the system reliability subject to various constraints regarding e.g. time, space, and energy. The applications of our achievements are two-fold: (i) the primary/backup approach technique, which is a fault tolerant one based on two task copies (primary and backup ones), and (ii) the CubeSat project, within framework of which small satellites operate in the harsh space environment. In both cases, the system performances are mainly evaluated by means of the rejection rate, the algorithm complexity measured by the number of comparisons of evaluated slots or the scheduling time, and the resilience assessed by injecting faults.

Primary/Backup Approach Technique

Our achievements for the primary/backup (PB) approach (presented and discussed in Chapter 3) include the introduction of a new processor allocation policy (called the *first found solution search: slot by slot*), and of three new enhancing techniques (the *restricted scheduling windows*, *limitation on the number of comparisons* and *several scheduling attempts*). We also present a mathematical programming formulation of the scheduling problem and carry out more general experiments to evaluate the fault tolerance of this approach.

The next five paragraphs enumerate the main results of the scheduling of the independent tasks. The first four paragraphs predominantly cover our achievements for the analysis of the main already existing methods for the PB approach. To the best of author's knowledge, although these methods are often put into practice, they have never been thoroughly analysed and compared. Our proposed enhancements for the PB approach are described in the fifth paragraph.

Firstly, the results of the PB approach in itself (considered as the baseline) and the one with backup copy (BC) overloading reveal that the BC overloading facilitates the reduction of the rejection rate (for example by 14% for a 14-processor system). When the BC deallocation is then put into practice, the improvement is even more noteworthy. For instance, for the 14-processor system, the gain is about 75% compared to the baseline PB approach and regardless of whether the BC overloading is implemented or not. Moreover, it is shown that the BC overloading and the BC deallocation well cooperate.

Secondly, we analysed the active PB approach, i.e. a technique allowing the primary and backup copies of the same task to be executed at the same time on different processors, which is not normally authorised. It is demonstrated that it is beneficial for systems dealing with tasks with tight deadline. It reduces the rejection rate compared to the baseline. For the 14-processor system, there is a drop in the rejection rate by about 17% for both the PB approach with BC deallocation and with or without BC overloading.

Thirdly, three different processor allocation policies were analysed. Although the *exhaustive search* (ES) exhibits lower rejection rate than both the *first found solution search - processor by processor* (FFSS PbP) and the *first found solution search - slot by slot* (FFSS SbS), its number of comparisons related to the algorithm run-time is significantly higher. The FFSS SbS performs better by all studied metrics than the FFSS PbP and it is 2-competitive in comparison to the optimal solution.

Fourthly, two scheduling search techniques were compared: the *free slot search technique* (FSST) and the *boundary schedule search technique* (BSST). The BBST + ES exhibits similar rejection rate as the FSST + ES while the number of comparisons of the BSST is significantly higher than the one of the

FSST (more than twice as large). Therefore, the BSST is not a convenient scheduling search technique to reduce the algorithm run-time.

And fifthly, three techniques (the *limitation on the number of comparisons*, *restricted scheduling windows* and *several scheduling attempts*) and their combinations are analysed in terms of their performances. The results show that the best methods, which reduce both the rejection rate and the number of comparisons, are (i) the limitation on the number of comparisons combined with two scheduling attempts at 33% of the task window, and (ii) the limitation on the number of comparisons. The algorithm run-time of the former technique is reduced by 23% (mean value) and 67% (maximum value) and its rejection rate is decreased by 4% compared to the primary/backup approach without any enhancing method.

To extend the analysis of the PB approach of the independent tasks to the dependent ones, we adapted the previously studied scheduling algorithm. When the search for a slot to schedule a task copy is carried out by the BSST, the number of comparisons is significantly higher than the one based on the FSST. While the BSST scours all processors and tests all possible slots, the FSST conducts a search until a solution is found or all processors are tested. Consequently, the BSST + ES BC maxOverload, i.e. the method based on the BSST and maximising the BC overloading, exhibits better performances than other studied techniques in terms of the rejection rate and system throughout but at the cost of the longer algorithm run-time, except for the systems with only several processors. Furthermore, the FFSS SbS and FFSS PbP achieve similar performances but the FFSS SbS requires more comparisons.

Last but not least, simulations conducted for all presented algorithms unveil that the faults, having fault rates even higher than the worst estimated fault rate in a harsh environment ($1 \cdot 10^{-5}$ fault/ms [118]), have a minimal impact on the scheduling algorithm. Regarding the dependent tasks, the space and time constraints due to task dependencies impose more restrictions on scheduling than faults.

Although a considerable amount of work was carried out for the PB approach, it was mainly concerned with the system reliability. Therefore, we suggest as a possibility to follow up with the research taking into account also energy aspect, for power consumption is one of the most important issues in multiprocessor embedded systems. Another possibility is to consider that the real computation time may be shorter than the worst-case computation time.

In addition, our results for dependent tasks show that there is a room for further improvement. Nevertheless, to yield better results, it will be necessary to focus on particular applications instead of devising general methods as we did.

CubeSats

To make CubeSats fault tolerant, in Chapter 5, we propose to take advantage of multiprocessor architecture and gather all CubeSat processors on one board [1]. Such a multiprocessor system can make use of one of our devised algorithms, which schedules all tasks on board, detects faults and takes appropriate measures in order to deliver correct results.

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. While the first two presented algorithms do not take energy constraints into account, the last proposed algorithm called OneOffEnergy is energy-aware. All algorithms can use different ordering policies to sort a task queue. Overall, the "Earliest Deadline" ordering policy performs well in terms of the rejection rate and scheduling time for OneOff and the "Minimum Slack" ordering policy for OneOff&Cyclic.

All in all, the presented results based on two real CubeSat scenarios show that it is useless (from the viewpoint of the rejection rate) to consider systems with more than six processors and that OneOff

1. At present, every CubeSat system has in general one dedicated processor.

performs better than OneOff&Cyclic in terms of both 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. We therefore recommend that teams designing CubeSats gathering all processors on one board should make use of OneOff when choosing a no-energy-aware algorithm. Nevertheless, it would be better to apply an energy-aware algorithm, such as OneOffEnergy.

OneOffEnergy is a modified version of OneOff operating in two processor modes (Run and Standby) to save energy and it considers three system modes (normal, safe and critical) depending on the current energy stored in the battery. OneOffEnergy was not only evaluated for CubeSats but also in the context of another energy-constrained application.

The energy balance for CubeSat Scenario APSS showed that the communication phase requires a huge amount of energy mainly due to high power consumption of the transmitter. Compared to the CubeSat orbit duration taking 95 minutes, the communication phase lasts for 10 minutes but it may cause a lack of energy in the case if an energy-aware algorithm is not considered. If OneOffEnergy is implemented and a CubeSat operates within one of the system modes (normal, safe or critical), it does not take a risk to experience any energy shortage because the supplied energy covers all energy expenses.

Since the CubeSat scenario does not allow us to assess all performances of OneOffEnergy, we carried out simulations also for another energy-constrained application and compare the performances of OneOffEnergy with other simpler algorithms. The main differences are as follows: (i) the data for the application were synthetically generated (instead of real data for CubeSats), (ii) the time spent in the daylight and eclipse were shortened to reduce the duration of the simulation, and (iii) the battery capacity is not overestimated, as it is the case aboard CubeSats to avoid energy shortages.

The energy savings obtained when putting processors in Standby mode are not negligible and can avoid a lack of energy. Its use is thereby appropriate for systems with energy constraints because the system operates longer in normal or safe system modes and therefore avoid automatic rejection of low-priority tasks.

A system operating in normal mode only exhibits lower rejection rate than the one using OneOffEnergy but it is not able to run all the time due to limited energy resources. In contrast, OneOffEnergy, which checks the battery energy level and chooses a system mode accordingly, executes tasks with appropriate priorities to optimise the energy consumption and avoids energy shortages. Thus, our algorithm presents a reasonable trade-off between the system operation, such as the number of tasks and their priority, and the energy constraints.

Last but not least, all three devised algorithms (OneOff, OneOff&Cyclic and OneOffEnergy) were evaluated in a harsh environment and the results show that faults have a minimal impact on their performances up to $1 \cdot 10^{-4}$ fault/ms, which is higher than the worst estimated fault rate (10^{-5} fault/ms [118].

Although simulations to evaluate the proposed algorithm performances were carried out and analysed, the implementation on a real CubeSat platform might bring new interesting insights. In particular, the measurements of real power consumption and energy stored in the battery would be valuable benefits for further research.

Appendix A

Adaptation of the Boundary Schedule Search Technique to the First Found Solution Search: Processor by Processor

The method of *boundary schedules* was presented in [155] and described in Section 2.4.4. We remind the reader that boundary "schedules" are slots having their start time and/or finish time at the same time as boundaries of already scheduled task copies.

The *boundary schedule search technique* (BSST) is mainly meant for the exhaustive search, which scours all processors to test all possibilities and to evaluate the overlap percentage among overloadable backup copies. In order to carry out comparisons with other scheduling techniques presented in Section 3.1.1, we realised several modifications to adapt this scheduling search also to the non-exhaustive searches.

These modifications are described in this appendix. First, it presents the scheduling of primary copies and then the one of backup copies. Only modifications for the FFSS PbP are considered and were realised here because the modifications for the FFSS SbS would require even higher scheduling control.

We remind the reader that all assumptions formulated in Section 3.1.1 remind valid.

A.1 Primary Copies

The primary copies are scheduled as soon as possible. They can start at their arrival time, which is considered as an "imaginary" boundary, or at the end of an already placed copy if the corresponding free slot is large enough. Thus, there is no difference in terms of the search for a slot between the ES and the FFSS.

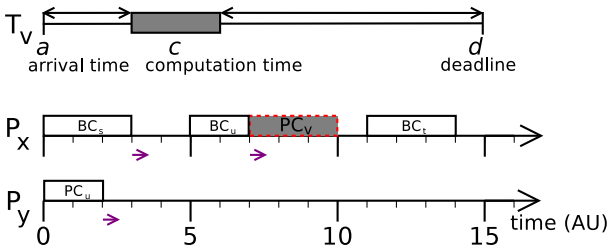


Figure A.1 – Example of the search for a PC slot using the BSST + FFSS PbP

Figure A.1 depicts an example of the search for a slot for the primary copy of task T_v using the BSST + FFSS PbP. Every possible attempt to schedule a copy starting/ending at a given boundary is illustrated using a violet arrow, which also indicates its direction. In this case, the primary copy PC_v is scheduled after two attempts on processor P_x and the processor P_y is consequently not treated. This

example shows that a slot can be found more quickly and with less complexity compared to the exhaustive search but at the cost of missing better slots, such as the slot for PC_v on processor P_y starting earlier than on processor P_x.

A.2 Backup Copies

Regarding that the BSST was not originally meant for the non-exhaustive search, we made several modifications for scheduling of the backup copies. Therefore, to replace the computation of the overlap percentage among overloadable backup copies, we introduce special rules aiming at maximising the BC overloading, which leads to the non-sequential search and thus higher system control. The earliest time when a backup copy can start its execution, i.e. when a primary copy finishes its execution, denoted by s, and the task deadline d may also be considered as boundaries. If scheduling the backup copies, we distinguish two cases according to whether the BC overloading is authorised or not.

A.2.1 No BC Overloading

If the BC overloading is not authorised, the backup copy is in general scheduled as late as possible.

The task deadline d is not considered as a boundary, if there is no copy at all within the scheduling window in which case a backup copy is placed on the left of the deadline, as shown in Figure A.3a. If the algorithm searches for a BC slot, it checks all slots on the left of start boundaries of existing copies within the scheduling window. Except for the last free slot, when the algorithm verifies the slot on the right of the end boundary.

On the one hand, a merit of the BSST + FFSS PbP is that the last free slot, i.e. one containing d, within the scheduling window is not divided into two free slots, which contributes to form clusters of task copies and to avoid creating two smaller free slots. On the other hand, this benefit is also a drawback because in the last free slot, a backup copy is generally scheduled in the left part of the free slot and therefore not as late as possible.

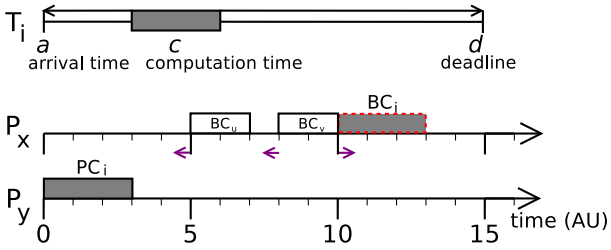
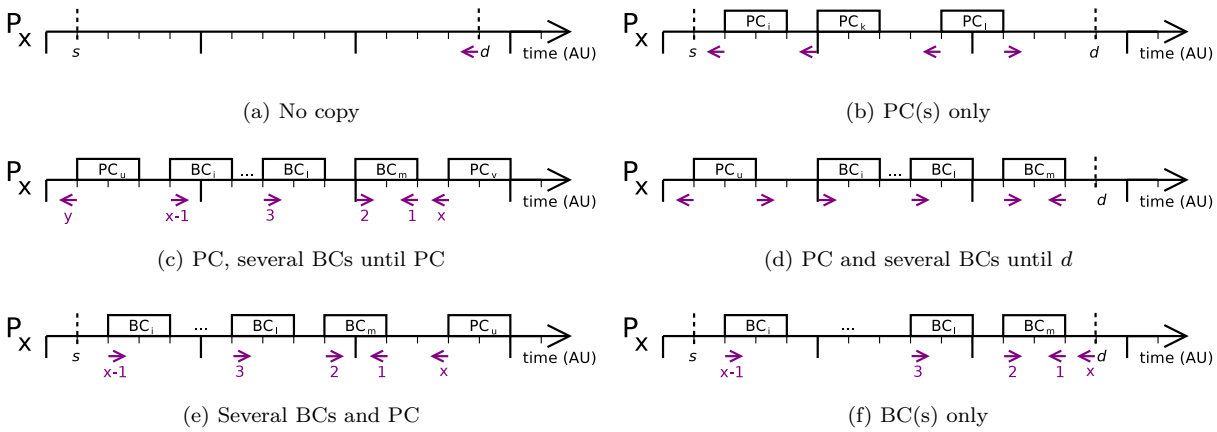


Figure A.2 – Example of search for a slot for BC

An example of BC scheduling is represented in Figure A.2. Although there are three slots (indicated by purple arrows), where a BC of task T_v can be scheduled, the BSST + FFSS PbP does not test all of them because the backup copy BC_v can be scheduled after the first attempt.

A.2.2 BC Overloading Authorised

When the BC overloading is authorised, the algorithm becomes more complex, for more tests of boundary slots are required compared to the approach without BC overloading. In general, the aim of this technique is to maximise the overlap of overloadable backup copies. Recall that two backup copies having their respective primary copies on the same processor cannot overload each other. This is the reason why our algorithm, usually searching from the last free slot to the first available time for a BC slot, requires to return back to test slots having later start time but smaller value of the overlap percentage.



(a) No copy

(b) PC(s) only

(c) PC, several BCs until PC

(d) PC and several BCs until d

(e) Several BCs and PC

(f) BC(s) only

Figure A.3 – Different cases of BC scheduling with BC overloading

In this case, we consider that the start s of the BC scheduling window is never considered as a boundary and that the task deadline d is considered as a boundary:

— if and only if there is no copy at all within the scheduling window and in this case, depicted in Figure A.3a, the algorithm immediately tests the slot on the left;

— if and only if there is no primary copy within the scheduling window, as shown in Figure A.3f, and in this situation it first waits after all backup copies are tested and then, if no slot is found, it tests the slot on the left.

As the BC overloading is authorised, the algorithm distinguishes for each current copy whether a boundary belongs to a primary or backup copy, as it was shown in Figures 3.4.

If a current copy is a primary copy, its end boundary is not tested if there is no copy (Figure A.3b) or no primary copy (Figure A.3d) between the end boundary and d. The algorithm checks the slot on the right of the boundary.

When the algorithm encounters a start boundary of primary copy, it tests the slot on the left. This verification is carried out:

— immediately, if the previous copy is a primary copy (Figure A.3b) or if there is no copy on the left of the start boundary within the scheduling window;

— after checking all backup copies until previous PC and no slot is found, if the previous copies are the backup ones on the left of the start boundary within scheduling window (Figures A.3c and A.3e).

The second case is when a current copy is a backup copy. Its start boundary is always taken into account and the algorithm checks the slot on the right (for instance BC_i or BC_l in Figures A.3c, A.3d, A.3e and A.3f). The end boundary is not used unless this copy is the first backup copy tested in the current free slot (for example BC_m in Figures A.3c, A.3d, A.3e and A.3f) in which case the algorithm verifies the slot on the left.

An idea to simplify the algorithm when the BC overloading is authorised is to consider only one boundary of backup copy (start or end one) and thus avoid several special cases.

Appendix B

DAGGEN Parameters

In Section 2.6.2, several task graph generators were briefly presented. The generator to generate directed acyclic graphs (DAGs) used in this thesis is DAGGEN [1]. The aim of this section is to illustrate the main parameters of this tool and show their influence on the DAG structure. Before presenting different parameters, we define *level* as it is shown in Figure B.1.

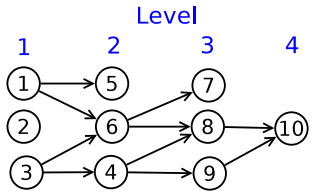
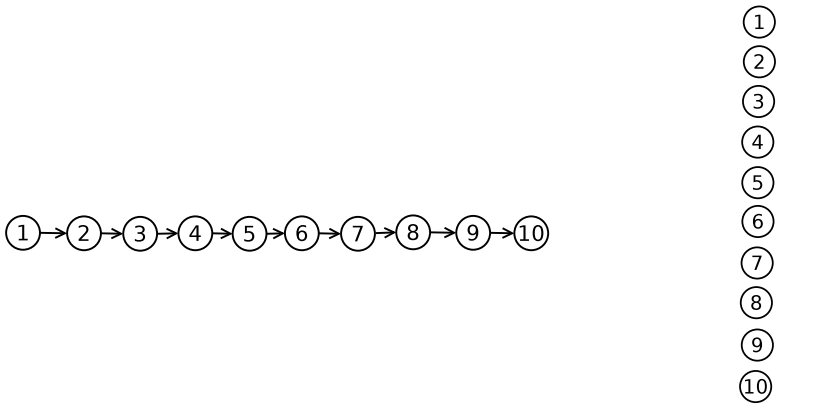


Figure B.1 – Levels of DAG

— Number of tasks or nodes = size *(1 value set by user)*
— Fat = width *(1 value set by user)*: Figure B.2
This parameter denotes the maximum number of tasks that can be executed concurrently. If it is equal to 0.0, we get "chain" graphs with minimum parallelism, while if the value is set at 1.0 there are "fork-join" graphs with maximum parallelism.

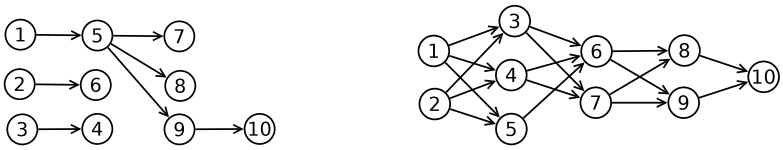


(a) Fat = 0.0 (b) Fat = 1.0

Figure B.2 – Example of DAG parameter "fat"

— Density *(1 value set by user)*: Figure B.3
Density is the number of edges between two levels of the DAG. If it is set at 0.0, a DAG has only a few edges, which means minimum dependencies. If it is equal to 1.0, a DAG is a full graph with many edges.

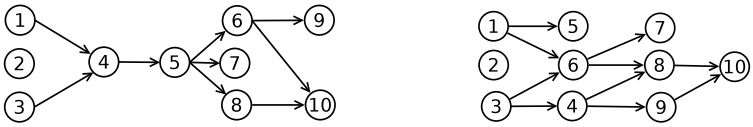
1. https://github.com/frs69wq/daggen



(a) Density = 0.0 (b) Density = 1.0

Figure B.3 – Example of DAG parameter "density"

— Regularity *(1 value set by user)*: Figure B.4
Regularity determines the uniformity of the number of tasks in each level. A DAG is irregular if this parameter is set at 0 and perfectly regular if it is equal to 1.



(a) Regularity = 0.0 (b) Regularity = 1.0

Figure B.4 – Example of DAG parameter "regularity"

— Jump *(1 value set by user)*: Figure B.5
This parameter determines the number of levels spanned by communications, i.e. random edges going from level l to level $l + jump$. If $jump = 1$, there is no jumping "over" any level.

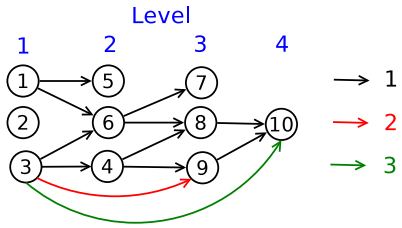


Figure B.5 – Example of DAG parameter "jump"

— Data size *(min and max values set by user)*
This parameter denotes the size of data processed by a task.
— Extra parameter *(min and max values set by user)*
An example of the extra parameter is the Amdahl's law parameter, which represents an overhead of parallelization of tasks in parallel task graphs.
— Communication (MBytes) to computation (sec) ratio *(1 value to be chosen)*
This ratio encodes the complexity of the computation of a task depending on the number of elements n in the dataset if processes. One of the following formula can be chosen ($a \in [26; 29]$):
  — $a \cdot n$
  — $a \cdot n \log(n)$
  — $n^{3/2}$

Appendix C

Constraint Programming Parameters

This appendix deals with the settings of some simulation parameters in CPLEX optimiser [1] when solving constraint programming (CP) problems. Actually, when resolving the same problem on two different computational resources, the result may differ because the setting parameters using the default values are platform-dependent. Consequently, the model can choose a different solution when there are several alternate optimal solutions to the problem. In order to reproduce the same results, these parameters need to be set properly [77].

Table C.1 sums up the parameters that were studied during the thesis and have influence on the reproducibility of the results. While the default settings of parameters `FailLimit` and `TimeLimit` are independent of the computational resources, the default setting of the parameter `Workers` is platform-dependent because its number equals the number of central processing units (CPUs) available.

Table C.1 – Several constraint programming (CP) setting parameters [74, 75, 76]

Parameter	Definition	Default value
`FailLimit`	Limits the number of failures [2] that can occur before terminating the search	2 100 000 000
`TimeLimit`	Limits the CPU time spent solving before terminating a search	Infinity (s) $= 1 \cdot e^{+75}\ s$
`Workers`	Sets the number of workers to run in parallel to solve the model	Automatic (as many workers as there are CPUs available)

As an example of the influence of the last mentioned parameter, we consider a CubeSat scenario and the task input consisting of 500 independent tasks. Each task has three copies and dynamically arrives on a 5-processor system. This task data set, which remained exactly the same in all experiments, was executed on a computer composed of four CPUs and on a computing platform equipped with twelve CPUs, while we varied the parameters `TimeLimit`, `FailLimit` and `Workers`. The mathematical formulation of the analysed problem is described in Section 5.2.2.1, whose objective function is to maximise the number of accepted tasks and therefore to minimise the rejection rate.

For every experiment, we measured the time elapsed to find an optimal solution and we recorded the rejection rate and processor load of the given solution. The obtained results are encapsulated in Table C.2.

First and foremost, we note that, in order to obtain the same result, the parameter `Workers` needs to be set at the same value. For example, if `TimeLimit` =Infinity, `FailLimit` = 10 000 and `Workers` = 4, the optimal solution is the same, as highlighted in red.

While it is true for the computing platform that the higher the parameter `FailLimit`, the lower the rejection rate, it is not the case for the computer because the rejection rate when `FailLimit` = 10 000 is higher than when `FailLimit` = 5 000. It can be also seen that the higher the value of `FailLimit`,

1. `https://www.ibm.com/analytics/cplex-optimizer`
2. The number of failures or the number of fails stands for the number of branches explored in the binary search tree, which did not lead to a solution [73].

Table C.2 – Example of the influence of parameter settings

`TimeLimit`	`FailLimit`	**Computer** (# CPUs = 4)				**Computing platform** (# CPUs = 12)			
		# workers	Rejection rate	Processor load	Duration (min)	# workers	Rejection rate	Processor load	Duration (min)
Infinity	5 000	4	0.166	0.876	162	12	0.476	0.538	33.7
Infinity	10 000	4	0.172	0.847	130	4	0.172	0.847	43.7
Infinity	10 000					12	0.112	0.923	44.6
Infinity	15 000	4	0.104	0.927	164	12	0.112	0.923	47.3
Infinity	20 000	4	0.104	0.927	234	12	0.112	0.923	47.3

the longer the time elapsed to find a solution (even though the increase depends on the computational resources).

We also try not to limit `FailLimit`. In this case, a CPLEX optimiser treated only the first 22 arrived tasks within the first 30 minutes of simulation (no matter whether a computer or computing platform was used). The simulations were then stopped because the duration exponentially increases with the number of tasks and the time to find an optimal solution would be too long if a simulation would finish at all.

Therefore, the value of `FailLimit` to be chosen for simulations is the value that corresponds to the first case when studied metrics do not vary any more when the parameter `FailLimit` increases on a given computational resource (marked by green and blue in Table C.2). In the studied case, `FailLimit` equals 10 000 for the computing platform and 15 000 for the computer.

Furthermore, if the value of `FailLimit` is too low, the rejection rate may be very high, e.g. 0.476 for the computing platform with `FailLimit` = 5 000 and 12 workers.

We notice that it is useless to limit `TimeLimit` because presented results finish in reasonable time and that, although the resolutions are carried out faster when the number of CPUs increases, the higher number of CPUs available does not mean that the results will be better. In the analysed example, the rejection rate of the optimal solution of the computing platform (0.112) is higher by 7.7% compared to the one delivered by the computer (0.104). This is caused by a different decision made when several alternate optimal solutions to the problem are available [77].

Throughout this thesis, the parameter `TimeLimit` was kept to the default value and the parameter `FailLimit` was fixed at 10 000 for all conducted resolutions in CPLEX optimiser. While the problem optimisations for the primary/backup approach were realised on a computing platform equipped with 12 CPUs, the one for CubeSats were conducted on a computer composed of 4 CPUs.

Appendix D

Box Plot

The *box plot*, also known as the *box-and-whisker plot*, is a histogram-like method invented by John Wilder Tukey [43, 98, 113, 150]. This graphical tool is used to represent statistical data, in particular their location and variation information. An example of a box plot is depicted in Figure D.1. A diagram has two or three parts, which are as follows: box, whiskers and circle(s).

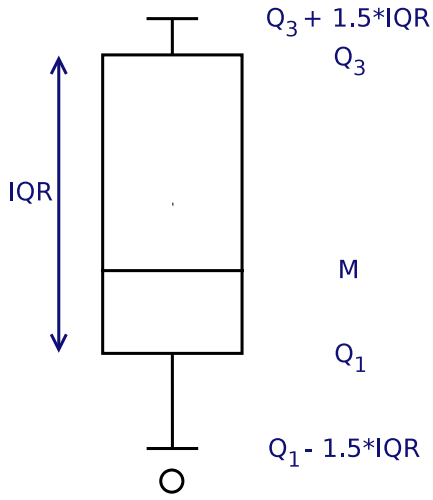


Figure D.1 – Example of a box plot

Before explaining the meaning of each part, we give several definitions.

— A *quartile* is one of the four divisions of data set and it splits this data set into four equal parts. The first quartile Q_1 is called *lower quartile value* and corresponds to the 25^{th} percentile. The third quartile, named *upper quartile value*, is the 75^{th} percentile.

— The *median* is the value of the point which is situated in the middle of the data set. It means that one half has the data smaller than this point and another half has the data larger than this point. If the number of data is odd, the median value is included in both halves.

— The *interquartile range* (IQR) is the difference between the third and the first quartiles, i.e. $IQR = Q_3 - Q_1$. This interval divides a data set into two groups of equal size at the median.

The **box** is respectively delimited by the first and third quartiles Q_1 and Q_3. A horizontal line in the box represents the statistical median M.

The **whiskers** start at the end of the box and extend to the outermost points that are not outliers, which means that they are within 1.5 times the interquartile range of Q_1 and Q_3.

The **circle** represents an outlier for a studied data set. An *outlier* is a value that is more than 1.5 times the interquartile range from the end of a box.

PUBLICATIONS

P. Dobiáš, E. Casseau, and O. Sinnen, *Restricted Scheduling Windows for Dynamic Fault-Tolerant Primary/Backup Approach-Based Scheduling on Embedded Systems*, in Proceedings of the 21st International Workshop on Software and Compilers for Embedded Systems, SCOPES '18, May 2018, pp. 27–30. https://doi.org/10.1145/3207719.3207724.

P. Dobiáš, E. Casseau, and O. Sinnen, *Comparison of Different Methods Making Use of Backup Copies for Fault-Tolerant Scheduling on Embedded Multiprocessor Systems*, in 2018 Conference on Design and Architectures for Signal and Image Processing (DASIP), Oct 2018, pp. 100–105. https://doi.org/10.1109/DASIP.2018.8597044.

P. Dobiáš, E. Casseau, and O. Sinnen, *Fault-Tolerant Online Scheduling Algorithms for CubeSats*, in Proceedings of the 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'2020, January 2020, pp. 1–6. https://doi.org/10.1145/3381427.3381430.

P. Dobiáš, E. Casseau, and O. Sinnen, *Evaluation of Fault Tolerant Online Scheduling Algorithms for CubeSats*, in Proceedings of the 23rd Euromicro Conference on Digital System Design, DSD'2020, August 2020, pp. 622–629. https://doi.org/10.1109/DSD51259.2020.00102.

BIBLIOGRAPHY

[1] *Live Real Time Satellite Tracking and Predictions*. https://www.n2yo.com/.

[2] *Documentation of AAU-Cubesat On Board Computer Software*, 2002. Aalborg University http://www.space.aau.dk/cubesat/dokumenter/software.pdf.

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

[4] K. Ahn, J. Kim, and S. Hong, *Fault-tolerant real-time scheduling using passive replicas*, in Proceedings Pacific Rim International Symposium on Fault-Tolerant Systems, Dec 1997, pp. 98–103. https://doi.org/10.1109/PRFTS.1997.640132.

[5] R. Al-Omari, A. K. Somani, and G. Manimaran, *A New Fault-Tolerant Technique for Improving Schedulability in Multiprocessor Real-Time Systems*, in Proceedings 15th International Parallel and Distributed Processing Symposium (IDPS), 2001. https://doi.org/10.1109/IPDPS.2001.924967.

[6] R. Al-Omari, A. K. Somani, and G. Manimaran, *Efficient Overloading Techniques for Primary-Backup Scheduling in Real-Time Systems*, in Journal of Parallel and Distributed Computing, vol. 64, 2004, pp. 629–648. https://doi.org/10.1016/j.jpdc.2004.03.015.

[7] S. Al-Sharaeh and B. E. Wells, *A Comparison of Heuristics for List Schedules using the Box-Method and P-Method for Random Digraph Generation*, in Proceedings of 28th Southeastern Symposium on System Theory, 1996, pp. 467–471. https://doi.org/10.1109/SSST.1996.493549.

[8] V. Almonacid and L. Franck, *Extending the Coverage of the Internet of Things with Low-Cost Nanosatellite Networks*, in Acta Astronautica, vol. 138, 2017, pp. 95–101. https://doi.org/10.1016/j.actaastro.2017.05.030.

[9] A. Amin, R. Ammar, and A. El Dessouly, *Scheduling real time parallel structures on cluster computing with possible processor failures*, in Proceedings. ISCC 2004. Ninth International Symposium on Computers And Communications (IEEE Cat. No.04TH8769), vol. 1, July 2004, pp. 62–67. https://doi.org/10.1109/ISCC.2004.1358382.

[10] K. Anderson, *Low-Cost, Radiation-Tolerant, On-Board Processing Solution*, in IEEE Aerospace Conference, March 2005, pp. 1–8. https://doi.org/10.1109/AERO.2005.1559533.

[11] K. Antonini, M. Langer, A. Farid, and U. Walter, *SWEET CubeSat – Water Detection and Water Quality Monitoring for the 21st Century*, in Acta Astronautica, vol. 140, 2017, pp. 10–17. https://doi.org/10.1016/j.actaastro.2017.07.046.

[12] G. E. Apostolakis, *Engineering Risk Benefit Analysis: Probability Distributions in RPRA*, 2007. Massachusetts Institute of Technology, https://ocw.mit.edu/courses/engineering-systems-division/esd-72-engineering-risk-benefit-analysis-spring-2007/lecture-notes/rpra3.pdf.

[13] 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.

[14] M. H. Arnesen and C. E. Kiær, *Mission Event Planning & Error-Recovery for CubeSat Applications*, Master's thesis, Norwegian University of Science and Technology, Department of Electronics and Telecommunications, 2014. http://hdl.handle.net/11250/2371107.

[15] G. Aupy, Y. Robert, and F. Vivien, *Assuming Failure Independence: Are We Right to be Wrong?*, in IEEE International Conference on Cluster Computing (CLUSTER), Sep 2017, pp. 709–716. https://doi.org/10.1109/CLUSTER.2017.24.

[16] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, *Basic Concepts and Taxonomy of Dependable and Secure Computing*, in IEEE Transactions on Dependable and Secure Computing, vol. 1, Jan 2004, pp. 11–33. https://doi.org/10.1109/TDSC.2004.2.

[17] H. Aysan, R. Dobrin, S. Punnekkat, and J. Proenza, *Probabilistic Scheduling Guarantees in Distributed Real-Time Systems under Error Bursts*, in Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies Factory Automation (ETFA 2012), Sep. 2012, pp. 1–9. https://doi.org/10.1109/ETFA.2012.6489644.

[18] J. Balasangameshwara and N. Raju, *Performance-Driven Load Balancing with a Primary-Backup Approach for Computational Grids with Low Communication Cost and Replication Cost*, in IEEE Transactions on Computers, vol. 62, 2013, pp. 990–1003. https://doi.org/10.1109/TC.2012.44.

[19] P. Bartram, C. P. Bridges, D. Bowman, and G. Shirville, *Software Defined Radio Baseband Processing for ESA ESEO Mission*, in 2017 IEEE Aerospace Conference, March 2017, pp. 1–9. https://doi.org/10.1109/AERO.2017.7943952.

[20] L. Bautista-Gomez, A. Gainaru, S. Perarnau, D. Tiwari, S. Gupta, C. Engelmann, F. Cappello, and M. Snir, *Reducing Waste in Extreme Scale Systems through Introspective Analysis*, in IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2016, pp. 212–221. https://doi.org/10.1109/IPDPS.2016.100.

[21] H. Beitollahi, S. G. Miremadi, and G. Deconinck, *Fault-Tolerant Earliest-Deadline-First Scheduling Algorithm*, in 2007 IEEE International Parallel and Distributed Processing Symposium, March 2007, pp. 1–6. https://doi.org/10.1109/IPDPS.2007.370608.

[22] J. K. Bekkeng, *Lecture on Radiation effects on space electronics.* Department of Physics, University of Oslo https://www.uio.no/studier/emner/matnat/fys/FYS4220/h11/undervisningsmateriale/forelesninger-vhdl/Radiation%20effects%20on%20space%20electronics.pdf.

[23] I. Benson, A. Kaplan, J. Flynn, and S. Katz, *Fault-Tolerant and Deterministic Flight-Software System For a High Performance CubeSat*, in International Journal of Grid and High Performance Computing (IJGHPC), vol. 9, 2017. https://doi.org/10.4018/IJGHPC.2017010108.

[24] P. Bernardi, L. M. V. Bolzani, M. Rebaudengo, M. S. Reorda, F. L. Vargas, and M. Violante, *A New Hybrid Fault Detection Technique for Systems-on-a-Chip*, in IEEE Transactions on Computers, vol. 55, Feb 2006, pp. 185–198. https://doi.org/10.1109/TC.2006.15.

[25] V. Berten, J. Goossens, and E. Jeannot, *A probabilistic approach for fault tolerant multiprocessor real-time scheduling*, in Proceedings 20th IEEE International Parallel Distributed Processing Symposium, April 2006. https://doi.org/10.1109/IPDPS.2006.1639409.

[26] A. A. Bertossi, L. V. Mancini, and F. Rossini, *Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems*, in IEEE Transactions on Parallel and Distributed Systems, vol. 10, Sep 1999, pp. 934–945. https://doi.org/10.1109/71.798317.

[27] T. Bleier, P. Clarke, J. Cutler, L. D. Martini, C. Dunson, S. Flagg, A. Lorenz, and E. Tapio, *QuakeSat Lessons Learned: Notes from the Development of a Triple CubeSat*, tech. rep., 2014. https://www.quakefinder.com/pdf/Lessons_Learned_Final.pdf.

[28] E. Braegen, D. Hayward, G. Hynd, and A. Thomas, *AdeSat: The Design and Build of a Small Satellite Based on CubeSat Standards (Final Report Level IV Honours)*, tech. rep., University of Adelaide, Australia, 2007.

[29] F. Bräuer, *System Architecture Definition of the DelFFi Command and Data Handling Subsystem* , Master's thesis, Faculty of Aerospace Engineering, Delf University of Technology, 2015. https://repository.tudelft.nl/islandora/object/uuid%3Afd8a851b-8e08-4560-a257-e9b17210de25.

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

[31] D. Burlyaev and R. van Leuken, *System Fault-Tolerance Analysis of COTS-based Satellite On-Board Computers*, in Microelectronics Journal, vol. 45, 2014, pp. 1335–1341. https://doi.org/10.1016/j.mejo.2014.01.007.

[32] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright, *Probabilistic Scheduling Guarantees for Fault-Tolerant Real-Time Systems*, in Dependable Computing for Critical Applications 7, Jan 1999, pp. 361–378. https://doi.org/10.1109/DCFTS.1999.814306.

[33] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Springer, 2011. https://doi.org/10.1007/978-1-4614-0676-1.

[34] A. Campbell, P. McDonald, and K. Ray, *Single event upset rates in space*, in IEEE Transactions on Nuclear Science, vol. 39, Dec 1992, pp. 1828–1835. https://doi.org/10.1109/23.211373.

[35] L. Chang, *Microsatellite Design and Integration for INSPIRE: International Satellite Project in Research and Education.* Presentation at APSCO & ISSI-BJ Space Science School on October 25, 2016, 2016. https://docplayer.net/144761913-Microsatellite-design-and-integration-for-inspire-international-satellite-project-html.

[36] 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.

[37] N. Chronas, *Gsoc project*, 2017. https://nchronas.github.io/GSoC-2017/.

[38] T. B. Clausen, A. Hedegaard, K. B. Rasmussen, R. L. Olsen, J. Lundkvist, and P. E. Nielsen, *Designing On Board Computer and Payload for the AAU CubeSat.* http://www.space.aau.dk/cubesat/dokumenter/article.pdf.

[39] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, *Random Graph Generation for Scheduling Simulations*, in Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, SIMUTools '10, ICST, Brussels, Belgium, Belgium, 2010, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 60:1–60:10. http://dl.acm.org/citation.cfm?id=1808143.1808219.

[40] D. Crettaz, *Control & Data Management System*, tech. rep., HES-SO, Sion, Switzerland, 2007. http://escgesrv1.epfl.ch/04%20-%20Command%20and%20data%20management/S3-C-CDMS-Report%20and%20tests.pdf.

[41] A. Das, A. Kumar, B. Veeravalli, C. Bolchini, and A. Miele, *Combined DVFS and Mapping Exploration for Lifetime and Soft-Error Susceptibility improvement in MPSoCs*, in Design, Automation Test in Europe Conference Exhibition (DATE), March 2014, pp. 1–6. https://doi.org/10.7873/DATE.2014.074.

[42] R. Devaraj, A. Sarkar, and S. Biswas, *Fault-Tolerant Preemptive Aperiodic RT Scheduling by Supervisory Control of TDES on Multiprocessors*, in ACM Trans. Embed. Comput. Syst., vol. 16, New York, NY, USA, April 2017, ACM, pp. 87:1–87:25. https://doi.org/10.1145/3012278.

[43] P. Dobiáš, *Mapping and Scheduling of Applications/Tasks onto Homogeneous Faulty Processors*, Master's thesis, ENSSAT Lannion & Master of Research at ISTIC Rennes, Univ Rennes, IRISA, France, 2017.

[44] ——, *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.

[45] J. J. Dongarra, E. Jeannot, E. Saule, and Z. Shi, *Bi-objective Scheduling Algorithms for Optimizing Makespan and Reliability on Heterogeneous Systems*, in Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '07, ACM, 2007, pp. 280–288. http://doi.acm.org/10.1145/1248377.1248423.

[46] G. Dósa and Y. He, *Semi-Online Algorithms for Parallel Machine Scheduling Problems*, in Computing, vol. 72, Jun 2004, pp. 355–363. https://doi.org/10.1007/s00607-003-0034-2.

[47] 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, March 2018, pp. 118–128. https://doi.org/10.1109/TR.2017.2765352.

[48] P. Duangmanee and P. Uthansakul, *Clock-Frequency Switching Technique for Energy Saving of Microcontroller Unit (MCU)-Based Sensor Node*, in Energies, vol. 11, 2018. https://doi.org/10.3390/en11051194.

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

[50] D. L. Dvorak, *AIAA Infotech@Aerospace Conference*, 2009, ch. NASA Study on Flight Software Complexity. https://arc.aiaa.org/doi/abs/10.2514/6.2009-1882.

[51] E. O. Elliott, *Estimates of Error Rates for Codes on Burst-Noise Channels*, in The Bell System Technical Journal, vol. 42, 1963, pp. 1977–1997. https://doi.org/10.1002/j.1538-7305.1963.tb00955.x.

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

[53] ——, *Nanosats Database*. https://airtable.com/shrafcwXODMMKeRgU/tbldJoOBP5wlNOJQY?blocks=hide.

[54] A. Erlank and C. Bridges, *Reliability Analysis of Multicellular System Architectures for Low-Cost Satellites*, in Acta Astronautica, vol. 147, 2018, pp. 183–194. https://doi.org/10.1016/j.actaastro.2018.04.006.

[55] 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. https://doi.org/10.1109/AERO.2017.7943732.

[56] 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.

[57] D. A. Galvan, B. Hemenway, W. W. IV, and D. Baiocchi, *Satellite Anomalies: Benefits of a Centralized Anomaly Database and Methods for Securely Sharing Information Among Satellite Operators*, tech. rep., RAND National Defense Research Institute, 2014. https://www.rand.org/pubs/research_reports/RR560.html#download.

[58] D. Geeroms, S. Bertho, M. De Roeve, R. Lempens, M. Ordies, and J. Prooth, *ARDUSAT, an Arduino-Based CubeSat Providing Students with the Opportunity to Create their own Satellite Experiment and Collect Real-World Space Data*, in 22nd ESA Symposium on European Rocket and Balloon Programmes and Related Research, L. Ouwehand, ed., vol. 730 of ESA Special Publication, Sep 2015, p. 643. https://ui.adsabs.harvard.edu/abs/2015ESASP.730..643G.

[59] S. Ghosh, R. Melhem, and D. Mosse, *Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System*, in Proceedings of 8th International Parallel Processing Symposium, April 1994, pp. 775–782. https://doi.org/10.1109/IPPS.1994.288216.

[60] S. Ghosh, R. Melhem, and D. Mosse, *Enhancing real-time schedules to tolerate transient faults*, in Proceedings 16th IEEE Real-Time Systems Symposium, Dec 1995, pp. 120–129. https://doi.org/10.1109/REAL.1995.495202.

[61] 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, March 1997, pp. 272–284. https://doi.org/10.1109/71.584093.

[62] E. N. Gilbert, *Capacity of a Burst-Noise Channel*, in The Bell System Technical Journal, vol. 39, 1960, pp. 1253–1265. https://doi.org/10.1002/j.1538-7305.1960.tb03959.x.

[63] B. Goel, S. A. McKee, and M. Själander, *Techniques to Measure, Model, and Manage Power*, vol. 87 of Advances in Computers, Elsevier, 2012, ch. 2, pp. 7–54. https://doi.org/10.1016/B978-0-12-396528-8.00002-X.

[64] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Soft-error Detection Using Control Flow Assertions*, in Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT'03), Nov 2003, pp. 581–588. https://doi.org/10.1109/DFTVS.2003.1250158.

[65] GomSpace, *NanoPower Battery 2600mAh Datasheet*, September 2019. Document No.: 1017178, https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanopower-battery_2600mAh.pdf.

[66] R. Graham, E. Lawler, J. Lenstra, and A. Kan, *Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey*, in Discrete Optimization II, P. Hammer, E. Johnson, and B. Korte, eds., vol. 5 of Annals of Discrete Mathematics, Elsevier, 1979, pp. 287–326. https://doi.org/10.1016/S0167-5060(08)70356-X.

[67] Y. Guo, D. Zhu, H. Aydin, J.-J. Han, and L. T. Yang, *Exploiting Primary/Backup Mechanism for Energy Efficiency in Dependable Real-Time Systems*, vol. 78, 2017, pp. 68–80. https://doi.org/10.1016/j.sysarc.2017.06.008.

[68] M. Hakem and F. Butelle, *Reliability and Scheduling on Systems Subject to Failures*, in International Conference on Parallel Processing, 2007. https://doi.org/10.1109/ICPP.2007.72.

[69] S. Hall. Team member of the RANGE CubeSat mission (Space Systems Design Lab, Georgia Institute of Technology), Private communication, 2019.

[70] L. Han, L. Canon, J. Liu, Y. Robert, and F. Vivien, *Improved Energy-Aware Strategies for Periodic Real-Time Tasks under Reliability Constraints*, in 2019 IEEE Real-Time Systems Symposium (RTSS), Dec 2019, pp. 17–29. https://doi.org/10.1109/RTSS46320.2019.00013.

[71] 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, March 2017, pp. 813–825. https://doi.org/10.1109/TPDS.2016.2600595.

[72] T. Herault and Y. Robert, *Fault-Tolerance Techniques for High-Performance Computing*, Springer Publishing Company, Incorporated, 1st ed., 2015. https://doi.org/10.1007/978-3-319-20943-2.

[73] IBM Knowledge Center, *Examining the engine log.* https://www.ibm.com/support/knowledgecenter/SSSA5P_12.10.0/ilog.odms.ide.help/OPL_Studio/usroplide/topics/opl_ide_stats_CP_exam_log.html.

[74] ———, *Search control/General options.* https://www.ibm.com/support/knowledgecenter/SSSA5P_12.10.0/ilog.odms.ide.help/OPL_Studio/oplparams/topics/opl_params_cpoptions_desc_search_general.html.

[75] ——, *Search control/Limits*. https://www.ibm.com/support/knowledgecenter/SSSA5P_12.10.0/ilog.odms.ide.help/OPL_Studio/oplparams/topics/opl_params_cpoptions_desc_search_limits.html.

[76] ——, *Setting CP parameters*. https://www.ibm.com/support/knowledgecenter/SSSA5P_12.10.0/ilog.odms.ide.help/OPL_Studio/opllanguser/topics/opl_languser_script_in_cp_params.html.

[77] IBM Support, *A note on reproducibility of CPLEX runs*. https://www.ibm.com/support/pages/node/397041.

[78] J. J. W. Howard and D. M. Hardage, *Spacecraft Environments Interactions: Space Radiation and Its Effects on Electronic Systems*, Tech. Rep. NASA/TP-1999-209373, National Aeronautics and Space Administration (NASA), 1999. https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19990116210.pdf.

[79] H. Jin, X. Sun, Z. Zheng, Z. Lan, and B. Xie, *Performance under Failures of DAG-based Parallel Computing*, in 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, May 2009, pp. 236–243. https://doi.org/10.1109/CCGRID.2009.55.

[80] A. Johnston and K. A. LaBel, *Single Event Effect Criticality Analysis: Effects in Electronic Devices and SEE Rates*, 1996. https://radhome.gsfc.nasa.gov/radhome/papers/seeca4.htm.

[81] 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. https://doi.org/10.1016/S0167-6377(98)00005-4.

[82] B. K. Kim, *Reliability analysis of real-time controllers with dual-modular temporal redundancy*, in Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306), Dec 1999, pp. 364–371. https://doi.org/10.1109/RTCSA.1999.811281.

[83] H. Kim, S. Lee, and B.-S. Jeong, *An improved feasible shortest path real-time fault-tolerant scheduling algorithm*, in Proceedings Seventh International Conference on Real-Time Computing Systems and Applications, Dec 2000, pp. 363–367. https://doi.org/10.1109/RTCSA.2000.896412.

[84] J. R. Kopacz, R. Herschitz, and J. Roney, *Small Satellites an Overview and Assessment*, in Acta Astronautica, vol. 170, 2020, pp. 93–105. https://doi.org/10.1016/j.actaastro.2020.01.034.

[85] 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.

[86] C. M. Krishna and K. G. Shin, *On Scheduling Tasks with a Quick Recovery from Failure*, in IEEE Transactions on Computers, vol. C-35, May 1986, pp. 448–455. https://doi.org/10.1109/TC.1986.1676787.

[87] A. Kumar, S. Panda, S. K. Pani, V. Baghel, and A. Panda, *Aco and Ga Based Fault-Tolerant Scheduling of Real-Time Tasks on Multiprocessor Systems - A Comparative Study*, in IEEE 8th International Conference on Intelligent Systems and Control (ISCO), 2014, pp. 120–126. https://doi.org/10.1109/ISCO.2014.7103930.

[88] N. Kumar, J. Mayank, and A. Mondal, *Reliability Aware Energy Optimized Scheduling of Non-Preemptive Periodic Real-Time Tasks on Heterogeneous Multiprocessor System*, in IEEE Transactions on Parallel and Distributed Systems, vol. 31, April 2020, pp. 871–885. https://doi.org/10.1109/TPDS.2019.2950251.

[89] 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.

[90] K. Laizans, I. Sünter, K. Zalite, H. Kuuste, M. Valgur, K. Tarbe, V. Allik, G. Olentšenko, P. Laes, S. Lätt, and M. Noorma, *Design of the Fault Tolerant Command and Data Handling Subsystem for ESTCube-1* , in Proceedings of the Estonian Academy of Sciences, 2014, pp. 222–231. https://doi.org/10.3176/proc.2014.2S.03.

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

[92] M. Langer and J. Bouwmeester, *Reliability of CubeSats – Statistical Data, Developers' Beliefs and the Way Forward*, in 30th Annual AIAA/USU Conference on Small Satellites: Logan, United States, 2016. https://repository.tudelft.nl/islandora/object/uuid:4c6668ff-c994-467f-a6de-6518f209962e?collection=research.

[93] F. M. Lavey. Team member of the Auckland Programme for Space Systems (University of Auckland), Private communication, 2019.

[94] Y. Ling and Y. Ouyang, *Real-Time Fault-Tolerant Scheduling Algorithm for Distributed Computing Systems* , in Journal of Digital Information Management, vol. 10, Oct 2012. https://www.questia.com/library/journal/1G1-338892919/real-time-fault-tolerant-scheduling-algorithm-for.

[95] A. Łukasik and D. Roszkowski, *PW-Sat2: Critical Design Review: Mission Analysis Report*, tech. rep., The Faculty of Power and Aeronautical Engineering, Warsaw University of Technology, November 2016. https://pw-sat.pl/wp-content/uploads/2014/07/PW-Sat2-C-00.01-MA-CDR.pdf.

[96] G. Manimaran and C. S. R. Murthy, *A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and its Analysis*, in IEEE Transactions on Parallel and Distributed Systems, vol. 9, 1998, pp. 1137–1152. https://doi.org/10.1109/71.735960.

[97] L. Marchal, H. Nagy, B. Simon, and F. Vivien, *Parallel scheduling of DAGs under memory constraints*, Tech. Rep. RR-9108, LIP - ENS Lyon, October 2017. https://hal.inria.fr/hal-01620255v2.

[98] D. L. Massart, J. Smeyers-verbeke, X. C. A, and K. Schlesier, *PRACTICAL DATA HANDLING Visual Presentation of Data by Means of Box Plots*, in LC-GC Europe, vol. 18, 2005, pp. 215–218. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.169.9952.

[99] J. Mei, K. Li, X. Zhou, and K. Li, *Fault-Tolerant Dynamic Rescheduling for Heterogeneous Computing Systems*, in Journal of Grid Computing, vol. 13, 2015, pp. 507–525. https://doi.org/10.1007/s10723-015-9331-1.

[100] Mission Design Division, *Small Spacecraft Technology State of the Art*, tech. rep., National Aeronautics and Space Administration, Ames Research Center, Moffett Field, California, December 2015. https://www.nasa.gov/sites/default/files/atoms/files/small_spacecraft_technology_state_of_the_art_2015_tagged.pdf.

[101] D. Mosse, R. Melhem, and S. Ghosh, *Analysis of a Fault-Tolerant Multiprocessor Scheduling Algorithm*, in Proceedings of IEEE 24th International Symposium on Fault-Tolerant Computing, June 1994, pp. 16–25. https://doi.org/10.1109/FTCS.1994.315661.

[102] D. Mosse, R. Melhem, and Sunondo Ghosh, *A nonpreemptive real-time scheduler with recovery from transient faults and its implementation*, in IEEE Transactions on Software Engineering, vol. 29, Aug 2003, pp. 752–767. https://doi.org/10.1109/TSE.2003.1223648.

[103] 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.

[104] M. Naedele, *Fault-Tolerant Real-Time Scheduling under Execution Time Constraints*, Tech. Rep. 76, ETH Zurich, Computer Engineering and Networks Laboratory (TIK), CH-8092 Zurich, June 1999. https://doi.org/10.3929/ethz-a-004287366.

[105] A. Naithani, S. Eyerman, and L. Eeckhout, *Reliability-Aware Scheduling on Heterogeneous Multicore Processors*, in IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 397–408. https://doi.org/10.1109/HPCA.2017.12.

[106] ——, *Optimizing Soft Error Reliability Through Scheduling on Heterogeneous Multicore Processors*, in IEEE Transactions on Computers, vol. 67, 2018, pp. 830–846. https://doi.org/10.1109/TC.2017.2779480.

[107] NASA, *Space Radiation Effects on Electronic Components in Low-Earth Orbit*, 1999. https://llis.nasa.gov/lesson/824.

[108] 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.

[109] K. A. Nasuddin, M. Abdullah, and N. S. Abdul Hamid, *Characterization of the South Atlantic Anomaly*, in Nonlinear Processes in Geophysics, vol. 26, 2019, pp. 25–35. https://doi.org/10.5194/npg-26-25-2019.

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

[111] National Geographic, *Orbital Objects*, 2019. https://www.nationalgeographic.com/science/space/solar-system/orbital-objects/.

[112] C. Nieto-Peroy and M. R. Emami, *CubeSat Mission: From Design to Operation*, in Applied Sciences, vol. 9, 2019. https://doi.org/10.3390/app9153110.

[113] NIST/SEMATECH, *e-Handbook of Statistical Methods*. http://www.itl.nist.gov/div898/handbook/eda/section3/boxplot.htm.

[114] M. Noca, G. Roethlisberger, F. Jordan, N. Scheidegger, T. Choueiri, B. Cosandier, F. George, and R. Krpoun, *SwissCube Mission and System Overview*, tech. rep., UNINE/HES-SO/EPFL, Lausanne, Switzerland, 2008. http://escgesrv1.epfl.ch/01%20-%20Systems%20and%20mission%20documents/S3-C-SET-2-0-CDR%20Mission_System_Overview.pdf.

[115] M. A. Normann, *Hardware Review of an On Board Controller for a Cubesat*, tech. rep., Norwegian University of Science and Technology, Trondheim, 2015. http://nuts.cubesat.no/upload/2016/03/12/hardware_review_magne_normann.pdf.

[116] M. Orr and O. Sinnen, *Integrating Task Duplication in Optimal Task Scheduling With Communication Delays*, in IEEE Transactions on Parallel and Distributed Systems, vol. 31, Oct 2020, pp. 2277–2288. https://doi.org/10.1109/TPDS.2020.2989767.

[117] R. M. Pathan, *Scheduling Algorithms For Fault-TolerantReal-Time Systems*, PhD thesis, Chalmers University of Technology, Götebotg, Sweden, 2010. http://www.cse.chalmers.se/~risat/papers/LicentiateThesis.pdf.

[118] R. M. Pathan, *Real-Time Scheduling Algorithm for Safety-Critical Systems on Faulty Multicore Environments*, in Real-Time Systems, vol. 53, 2017, pp. 45–81. https://doi.org/10.1007/s11241-016-9258-z.

[119] M. L. Pinedo, *Scheduling – Theory, Algorithms, and Systems*, Springer, fifth ed., 2016.

[120] Python. https://docs.python.org/3.8/library/random.html.

[121] X. Qin and H. Jiang, *A Novel Fault-Tolerant Scheduling Algorithm for Precedence Constrained Tasks in Real-Time Heterogeneous Systems*, in Parallel Computing, vol. 32, 2006, pp. 331–356. https://doi.org/10.1016/j.parco.2006.06.006.

[122] X. Qin, H. Jiang, and D. R. Swanson, *An efficient fault-tolerant scheduling algorithm for real-time tasks with precedence constraints in heterogeneous systems*, in Proceedings International Conference on Parallel Processing, Aug 2002, pp. 360–368. https://doi.org/10.1109/ICPP.2002.1040892.

[123] Z. Quan, Z.-J. Wang, T. Ye, and S. Guo, *Task Scheduling for Energy Consumption Constrained Parallel Applications on Heterogeneous Computing Systems*, in IEEE Transactions on Parallel and Distributed Systems, vol. 31, May 2020, pp. 1165–1182. https://doi.org/10.1109/TPDS.2019.2959533.

[124] N. Rattenbury. Core team member of the Auckland Programme for Space Systems (University of Auckland), Private communication, 2019.

[125] Y. Robert and F. Vivien, *Introduction to Scheduling*, CRC Press, Inc., 1st ed., 2009.

[126] A. K. Samal, A. K. Dash, P. C. Jena, S. K. Pani, and S. Sha, *Bio-inspired Approach to Fault-Tolerant Scheduling of Real-Time Tasks on Multiprocessor - A Study*, in IEEE Power, Communication and Information Technology Conference (PCITC), 2015, pp. 905–911. https://doi.org/10.1109/PCITC.2015.7438125.

[127] G. Santilli, C. Vendittozzi, C. Cappelletti, S. Battistini, and P. Gessini, *CubeSat Constellations for Disaster Management in Remote Areas*, in Acta Astronautica, vol. 145, 2018, pp. 11–17. https://doi.org/10.1016/j.actaastro.2017.12.050.

[128] S. Sarkar, *Internet of Things—robustness and reliability*, Morgan Kaufmann, 2016, ch. 11, pp. 201–218. https://doi.org/10.1016/B978-0-12-805395-9.00011-3.

[129] A. Scholz, *Command and Data Handling System Design for the Compass-1 Picosatellite*, 2005. University of Applied Sciences Aachen http://www.raumfahrt.fh-aachen.de/compass-1/download/IAA-B5-0601_Abstract.pdf.

[130] B. Schroeder and G. A. Gibson, *A Large-Scale Study of Failures in High-Performance Computing Systems*, in IEEE Transactions on Dependable and Secure Computing, vol. 7, Oct 2010, pp. 337–350. https://doi.org/10.1109/TDSC.2009.4.

[131] M. Short and J. Proenza, *Towards Efficient Probabilistic Scheduling Guarantees for Real-Time Systems Subject to Random Errors and Random Bursts of Errors*, in 25th Euromicro Conference on Real-Time Systems, July 2013, pp. 259–268. https://doi.org/10.1109/ECRTS.2013.35.

[132] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, *Mapping on Multi/Many-core Systems: Survey of Current and Emerging Trends*, in 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), May 2013, pp. 1–10. https://doi.org/10.1145/2463209.2488734.

[133] 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.

[134] O. Sinnen, *Task Scheduling for Parallel Systems*, John Wiley & Sons, Ltd, 2007. https://doi.org/10.1002/0470121173.

[135] D. Spiers, *Chapter IIB-2 - Batteries in PV Systems*, in Practical Handbook of Photovoltaics, A. McEvoy, T. Markvart, and L. Castañer, eds., Academic Press, Boston, second edition ed., 2012, pp. 721–776. https://doi.org/10.1016/B978-0-12-385934-1.00022-2.

[136] R. Sridharan and R. Mahapatra, *Analysis of Real Time Embedded Applications in the Presence of a Stochastic Fault Model*, in 20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07), Jan 2007, pp. 83–88. https://doi.org/10.1109/VLSID.2007.36.

[137] STMicroelectronics, *STM32F103xF and STM32F103xG Datasheet*, May 2015. https://www.st.com/resource/en/datasheet/cd00253742.pdf.

[138] N. Stroud, *Evolving safety systems: Comparing lock-step, redundant execution and split-lock technologies*, 2018. https://community.arm.com/developer/ip-products/system/b/embedded-blog/posts/comparing-lock-step-redundant-execution-versus-split-lock-technologies.

[139] P. Struillou, *Probabilité : Support de cours*, 2015. ENSSAT Lannion.

[140] S. Stuijk, M. Geilen, and T. Basten, *SDF3: SDF For Free*, in Sixth International Conference on Application of Concurrency to System Design (ACSD'06), June 2006, pp. 276–278. https://doi.org/10.1109/ACSD.2006.23.

[141] G. Sulskus, *Lituanica SAT-1.* Presentation in July, 2014, 2014. https://ukamsat.files.wordpress.com/2014/07/lituanicasat-1-lo-78.pdf.

[142] W. Sun, Y. Zhang, C. Yu, X. Defago, and Y. Inoguchi, *Hybrid Overloading and Stochastic Analysis for Redundant Real-time Multiprocessor Systems*, in 2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007), Oct 2007, pp. 265–274. https://doi.org/10.1109/SRDS.2007.11.

[143] X. Tang, K. Li, R. Li, and B. Veeravalli, *Reliability-aware Scheduling Strategy for Heterogeneous Distributed Computing Systems*, in J. Parallel Distrib. Comput., vol. 70, Academic Press, Inc., Sept 2010, pp. 941–952. http://dx.doi.org/10.1016/j.jpdc.2010.05.002.

[144] 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.

[145] T. Tsuchiya, Y. Kakuda, and T. Kikuno, *Fault-tolerant scheduling algorithm for distributed real-time systems*, in Proceedings of Third Workshop on Parallel and Distributed Real-Time Systems, April 1995, pp. 99–103. https://doi.org/10.1109/WPDRTS.1995.470501.

[146] R. Vitali and M. G. Lutomski, *Derivation of Failure Rates and Probability of Failures for the International Space Station Probabilistic Risk Assessment Study*, in Probabilistic Safety Assessment and Management, C. Spitzer, U. Schmocker, and V. N. Dang, eds., Springer London, 2004, pp. 1194–1199. https://doi.org/10.1007/978-0-85729-410-4_193.

[147] I. Wali, *Circuit and System, Fault Tolerance Techniques*, PhD thesis, Université de Montpellier, 2016. https://tel.archives-ouvertes.fr/tel-01807927.

[148] 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, 03 2017, pp. 23–39. https://doi.org/10.1007/s10723-016-9386-7.

[149] G. Weerasinghe, I. Antonios, and L. Lipsky, *A generalized analytic performance model of distributed systems that perform N tasks using p fault-p*, in Proceedings 16th International Parallel and Distributed Processing Symposium, April 2002. https://doi.org/10.1109/IPDPS.2002.1016524.

[150] E. W. Weisstein, *Box-and-Whisker Plot, MathWorld (A Wolfram Web Resource).* http://mathworld.wolfram.com/Box-and-WhiskerPlot.html.

[151] ———, *Nondeterministic Turing Machine, MathWorld (A Wolfram Web Resource).* http://mathworld.wolfram.com/NondeterministicTuringMachine.html.

[152] ———, *NP-Problem, MathWorld (A Wolfram Web Resource).* http://mathworld.wolfram.com/NP-Problem.html.

[153] H. Xu, R. Li, C. Pan, and K. Li, *Minimizing Energy Consumption with Reliability Goal on Heterogeneous embedded Systems*, in Journal of Parallel and Distributed Computing, vol. 127, 2019, pp. 44–57. https://doi.org/10.1016/j.jpdc.2019.01.006.

[154] J. W. Young, *A First Order Approximation to the Optimum Checkpoint Interval*, in Commun. ACM, vol. 17, New York, NY, USA, Sep 1974, Association for Computing Machinery, pp. 530–531. https://doi.org/10.1145/361147.361115.

[155] 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, 2009, pp. 380–393. https://doi.org/10.1109/TC.2008.172.

[156] C. Zhu, Z. P. Gu, R. P. Dick, and L. Shang, *Reliable Multiprocessor System-on-chip Synthesis*, in Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS '07, New York, NY, USA, 2007, ACM, pp. 239–244. https://doi.org/10.1145/1289816.1289874.

[157] D. Zhu and H. Aydin, *Energy Management for Real-Time Embedded Systems with Reliability Requirements*, in 2006 IEEE/ACM International Conference on Computer Aided Design, Nov 2006, pp. 528–534. https://doi.org/10.1109/ICCAD.2006.320169.

[158] D. Zhu, R. Melhem, and D. Mosse, *The Effects of Energy Management on Reliability in Real-Time Embedded Systems*, in IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004., Nov 2004, pp. 35–40. https://doi.org/10.1109/ICCAD.2004.1382539.

[159] X. Zhu, X. Qin, and M. Qiu, *QoS-Aware Fault-Tolerant Scheduling for Real-Time Tasks on Heterogeneous Clusters*, in IEEE Transactions on Computers, vol. 60, June 2011, pp. 800–812. https://doi.org/10.1109/TC.2011.68.

[160] X. Zhu, J. Wang, H. Guo, D. Zhu, L. T. Yang, and L. Liu, *Fault-Tolerant Scheduling for Real-Time Scientific Workflows with Elastic Resource Provisioning in Virtualized Clouds*, in IEEE Transactions on Parallel and Distributed Systems, vol. 27, 2016, pp. 3501–3517. https://doi.org/10.1109/TPDS.2016.2543731.

[161] X. Zhu, J. Wang, J. Wang, and X. Qin, *Analysis and Design of Fault-Tolerant Scheduling for Real-Time Tasks on Earth-Observation Satellites*, in 43rd International Conference on Parallel Processing, 2014, pp. 491–500. https://doi.org/10.1109/ICPP.2014.58.

[162] A. Ünsal, B. Mumyakmaz, and N. Tunaboylu, *Predicting the Failures of Transformers in a Power System using the Poisson Distribution: A Case Study*, 12 2005. http://www.emo.org.tr/ekler/c22590152f4f53f_ek.pdf.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Contribution à l'ordonnancement dynamique, tolérant aux fautes, de tâches pour les systèmes embarqués temps-réel multiprocesseurs

Mot clés : Approche de "Primary/Backup", CubeSats, Multiprocesseurs, Placement dynamique, Systèmes embarqués temps réel, Tolérance aux fautes

Résumé : La thèse se focalise sur le placement et l'ordonnancement dynamique des tâches sur les systèmes embarqués multiprocesseurs pour améliorer leur fiabilité tout en tenant compte des contraintes telles que le temps réel ou l'énergie. Afin d'évaluer les performances du système, le nombre de tâches rejetées, la complexité de l'algorithme et la résilience estimée en injectant des fautes sont principalement analysés. La recherche est appliquée (i) à l'approche de « primary/backup » qui est une technique de tolérance aux fautes basée sur deux copies d'une tâche et (ii) aux algorithmes de placement pour les petits satellites appelés CubeSats.

Quant à l'approche de « primary/backup », l'objectif principal est d'étudier les stratégies d'allocation des processeurs, de proposer de nouvelles méthodes d'amélioration pour l'ordonnancement et d'en choisir une qui diminue considérablement la durée de l'exécution de l'algorithme sans dégrader les performances du système.

En ce qui concerne les CubeSats, l'idée est de regrouper tous les processeurs à bord et de concevoir des algorithmes d'ordonnancement afin de rendre les CubeSats plus robustes. Les scénarios provenant de deux CubeSats réels sont étudiés et les résultats montrent qu'il est inutile de considérer les systèmes ayant plus de six processeurs et que les algorithmes proposés fonctionnent bien même avec des capacités énergétiques limitées et dans un environnement hostile.

Title: Online Fault Tolerant Task Scheduling for Real-Time Multiprocessor Embedded Systems

Keywords: CubeSats, Fault Tolerance, Multiprocessors, Online Scheduling, Primary/Backup Approach, Real-Time Embedded Systems

Abstract: The thesis is concerned with online mapping and scheduling of tasks on multiprocessor embedded systems in order to improve the reliability subject to various constraints regarding e.g. time, or energy. To evaluate system performances, the number of rejected tasks, algorithm complexity and resilience assessed by injecting faults are analysed. The research was applied to: (i) the primary/backup approach technique, which is a fault tolerant one based on two task copies, and (ii) the scheduling algorithms for small satellites called CubeSats.

The chief objective for the primary/backup approach is to analyse processor allocation strategies, devise novel enhancing scheduling methods and to choose one, which significantly reduces the algorithm run-time without worsening the system performances.

Regarding CubeSats, the proposed idea is to gather all processors built into satellites on one board and design scheduling algorithms to make CubeSats more robust as to the faults. Two real CubeSat scenarios are analysed and it is found that it is useless to consider systems with more than six processors and that the presented algorithms perform well in a harsh environment and with energy constraints.