HAL
open science

Sûreté temporelle pour les systèmes temps réel multiprocesseurs

Frédéric Fauberteau

To cite this version:

Frédéric Fauberteau. Sûreté temporelle pour les systèmes temps réel multiprocesseurs. Algorithme et structure de données [cs.DS]. Université Paris-Est, 2011. Français. NNT : . tel-00668537v1

HAL Id: tel-00668537

https://theses.hal.science/tel-00668537v1

Submitted on 9 Feb 2012 (v1), last revised 1 Mar 2012 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ PARIS-EST

THÈSE

En vue d'obtenir le grade de
DOCTEUR DE L'UNIVERSITÉ PARIS-EST

Sûreté temporelle pour les systèmes temps réel multiprocesseurs

sous la direction de Gilles Roussel
et l'encadrement de Serge Midonnet

Spécialité **Informatique**
École doctorale Mathématiques et STIC (MSTIC) - ED 532
Soutenue publiquement par **Frédéric Fauberteau**
le **12 décembre 2011**

JURY :

Laurent	GEORGE	UPEC	Examinateur
Mathieu	JAN	CEA Saclay	Examinateur
Serge	MIDONNET	UPEMLV	Examinateur
Pascal	RICHARD	ENSMA	Rapporteur
Gilles	ROUSSEL	UPEMLV	Directeur
Yves	SOREL	INRIA Rocquencourt	Rapporteur

À Maryon mon amour...

Remerciements

Tout petit, on m'a appris à dire merci aux gens quand ils faisaient quelque chose pour vous...

Je tiens donc tout particulièrement à remercier Serge Midonnet pour m'avoir encadré durant toutes ces années. Un grand merci également à Laurent George pour avoir suivi mon travail de près. Merci à Gilles Roussel pour m'avoir permis de faire cette thèse. Et merci aussi à Mathieu Jan de me permettre de continuer à travailler sur des choses intéressantes. Toute ma gratitude va à Pascal Richard et à Yves Sorel pour avoir accepté de relire ce manuscrit, et pour leur commentaires avisés dans le but de l'améliorer.

Merci à Guillaume Blin d'avoir accepté d'être mon tuteur d'enseignement, et d'avoir souvent été de bon conseil. Merci à Line, à Séverine, à Pascale, à Marie-Hélène et à Gabrielle pour leur gentillesse, mais aussi pour tous les services rendus. Merci à tous les membres du LIGM, et à ceux de l'Esycom, ceux qui ont été mes enseignants, et les autres. Je ne citerai pas tous les noms, mais les visages me resteront en mémoire. Merci à tous pour avoir contribué à la bonne ambiance qui règne au 4° étage du bâtiment Copernic. Et puis merci aussi à Patrice, car même si la réponse était toujours « non », les conseils était souvent avisés.

Merci à Florian, mon compagnon de fortune, avec qui j'ai partagé le bureau pendant ces 3 années. Merci à Damien et à Manar, avec qui j'ai eu l'occasion de travailler en temps réel. Merci à Julien, Laurent, Elsa, Pierre, et tous les autres doctorants avec qui j'ai pu partager une discussion ou un repas. Et merci aussi aux membres du LaSTRE pour m'avoir si bien accueilli.

Merci également à mes parents, ma belle-famille et mes amis, qui ont toujours été présents.

J'ai peut-être beaucoup employé le mot « merci » sur cette page, mais ce ne sera jamais assez pour exprimer toute ma gratitude envers les personnes qui ont été là pour moi...

Résumé

Les systèmes temps réel à contraintes temporelles strictes sont caractérisés par des ensembles de tâches pour lesquelles sont connus l'échéance, le modèle d'arrivée (fréquence) et la durée d'exécution pire cas (WCET). Nous nous intéressons à l'ordonnancement de ces systèmes sur plate-forme multiprocesseur.

Garantir le respect des échéances pour un algorithme d'ordonnancement est l'une des problématiques majeures de cette thématique. Nous allons plus loin en nous intéressant à la *sûreté temporelle*, que nous caractérisons par les propriétés (i) de *robustesse* et (ii) de *viabilité*. La *robustesse* consiste à proposer un intervalle sur les augmentations (i-a) de WCET et (i-b) de fréquence tel que les échéances soient respectées. La *viabilité* consiste cette fois à garantir le respect des échéances lors du relâchement des contraintes (ii-a) de WCET (réduction), (ii-b) de fréquence (réduction) et (ii-c) d'échéance (augmentation). La *robustesse* revient alors à tolérer l'imprévu, tandis que la *viabilité* est la garantie que l'algorithme d'ordonnancement n'est pas sujet à des anomalies suite à un relâchement de contraintes.

Nous considérons l'ordonnancement en priorités fixes, où chaque occurrence d'une tâche est ordonnancée avec la même priorité. Dans un premier temps, nous étudions la propriété de robustesse dans les approches d'ordonnancement hors-ligne et sans migration (partitionnement). Nous traitons le cas des tâches avec ou sans partage de ressources. Dans un second temps, nous étudions la propriété de viabilité d'une approche d'ordonnancement en ligne avec migrations restreintes et sans partage de ressources.

Abstract

The hard real-time systems are characterized by sets of tasks for which are known the deadline, the arrival model (frequency) and the Worst-Case Execution Time (WCET). We focus on the scheduling of these systems on multiprocessor platforms.

One of the main issues of this topic is to ensure that all deadlines are met. We go further by focusing on the *temporal safety* which we characterized by the properties of (i) *robustness* and (ii) *sustainability*. The *robustness* consists in providing an interval on the increases of (i-a) WCET and (i-b) frequency in such a way that the deadlines are met. The *sustainability* consists in ensuring that no deadline is missed when the following constraints are relaxed : (ii-a) WCET (decreasing), (ii-b) frequency (decreasing) and (ii-c) deadline (increasing). The *robustness* amounts to tolerate unexpected behaviors while the *sustainability* is the guarantee that the scheduling algorithm does not suffer from anomalies because of a relaxation of constraints.

We consider fixed-priority scheduling for which any job of a task is scheduled with the same priority. Firstly, we study the property of robustness in off-line scheduling approaches without migration (partitioning). We deal with the case of tasks with or without shared resources. Secondly, we study the property of sustainability of an on-line restricted-migration scheduling approach without shared resources.

TABLE DES MATIÈRES

Remerciements **v**

Résumé **vii**

Abstract **ix**

Table des matières **xi**

Table des figures **xv**

Liste des tableaux **xvii**

1 Introduction générale **1**

1.1 Systèmes temps réel 1

1.2 Représentation des systèmes temps réel 3

1.2.1 Modèles de tâches 3

1.2.2 Temps de réponse 6

1.2.3 Modèles de processeurs 7

1.3 Ordonnancement temps réel 8

1.3.1 Ordonnancement hors-ligne / en ligne 8

1.3.2 Ordonnancement préemptif / non préemptif 8

1.3.3 Ordonnancement oisif / non oisif 9

1.3.4 Ordonnançabilité et faisabilité 9

1.3.5 Priorités dans l'ordonnancement 10

1.3.6 Attribution des priorités 12

1.4 Ordonnancement temps réel multiprocesseur 12

1.5 Motivations 15

1.6 Organisation 16

2 Sûreté temporelle **17**
2.1 Introduction . . . 17
2.2 Terminologie et notations . . . 19
2.3 Augmentation des paramètres temporels . . . 20
2.3.1 Marge sur les paramètres temporels . . . 21
2.3.2 Analyse de sensibilité . . . 23
2.4 Anomalies d'ordonnancement . . . 25
2.4.1 Prédictibilité de l'ordonnancement . . . 25
2.4.2 Viabilité de l'ordonnancement . . . 25
2.5 Conclusion . . . 27

3 Allocation de tâches indépendantes **29**
3.1 Introduction . . . 29
3.2 Notations . . . 30
3.3 Partitionnement . . . 30
3.3.1 Anomalie de partitionnement . . . 30
3.4 Algorithmes de partitionnement . . . 31
3.4.1 Algorithmes pour BIN-PACKING dans le cas du partitionnement . 32
3.4.2 Algorithmes modifiés . . . 34
3.4.3 Ordre sur les tâches . . . 35
3.4.4 Travaux existants . . . 36
3.5 Partitionnement robuste . . . 37
3.5.1 Description de l'algorithme . . . 38
3.5.2 Protocole de simulation . . . 38
3.5.3 Marge sur le WCET . . . 39
3.5.4 Marge sur la période . . . 42
3.6 Conclusion . . . 43

4 Allocation de tâches dépendantes **45**
4.1 Introduction . . . 45
4.2 Terminologie et notations . . . 46
4.2.1 Modèle . . . 46
4.2.2 Notations . . . 47
4.3 Protocoles de synchronisation . . . 47
4.3.1 MPCP . . . 47
4.3.2 MSRP . . . 48
4.3.3 FMLP . . . 48
4.4 Partitionnement de tâches dépendantes . . . 49
4.5 Partitionnement robuste . . . 50
4.5.1 Recuit simulé . . . 51
4.5.2 Algorithme de partitionnement . . . 51

4.5.3 Initialisation et voisinage aléatoire 52
4.5.4 Calcul de l'énergie 53
4.6 Évaluation 54
4.6.1 Protocole de simulation 54
4.6.2 Marge sur le WCET 54
4.6.3 Marge sur la fréquence 54
4.7 Conclusion 55

5 Ordonnancement à migrations restreintes 57
5.1 Introduction 57
5.2 Terminologie 58
5.3 Algorithme d'ordonnancement 59
5.3.1 Travaux existants 59
5.3.2 Laxité 59
5.3.3 Anomalies d'ordonnancement 60
5.3.4 Description de l'algorithme 61
5.4 Viabilité 64
5.4.1 Résistance aux diminutions de coûts d'exécution 64
5.4.2 Résistance aux augmentations d'échéances 66
5.4.3 Résistance aux augmentations de périodes 66
5.5 Tests d'ordonnançabilité 67
5.5.1 Test nécessaire et suffisant 67
5.5.2 Test suffisant 70
5.6 Conclusion 72

6 Simulateur 75
6.1 Introduction 75
6.2 Propriétés du logiciel 75
6.2.1 Licence 75
6.2.2 Langage de programmation 76
6.2.3 Interface utilisateur 76
6.2.4 Format des données 76
6.2.5 Validation 77
6.3 Architecture 77
6.3.1 Génération de tâches 77
6.3.2 Algorithmes pour la robustesse 78
6.3.3 Algorithmes de partitionnement 79
6.3.4 Algorithmes d'ordonnancement 80
6.4 Conclusion 81

7 Conclusion générale 83

Bibliographie **85**

Glossaire **97**

Acronymes **99**

Annexe 103

A Protocoles de synchronisation **103**

A.1 MPCP 103
A.2 MSRP 105
A.3 FMLP 106
A.3.1 Exemple 106
A.3.2 Pire temps de blocage 107

TABLE DES FIGURES

2.1 Algorithme ne supportant pas les variations négatives sur les paramètres. 18

3.1 Algorithmes de partitionnement avec utilisation décroissante, ordonnancement Deadline-Monotonic et analyse de temps de réponse. 39
3.2 Algorithmes de partitionnement avec Deadline-Monotonic et analyse de temps de réponse. Comparaison entre utilisation décroissante et laxité croissante. 41
3.3 Algorithmes de partitionnement avec utilisation décroissante, ordonnancement Deadline-Monotonic et analyse de temps de réponse. 42

4.1 Comparaison entre RPSA et SPA pour la marge sur le WCET. 54
4.2 Comparaison entre RPSA et SPA pour la marge sur les fréquences. . . . 55

5.1 Exemple de la non prédictibilité d'un algorithme d'ordonnancement à migrations restreintes [HL94]. 61
5.2 Exemple de non prédictibilité en migrations restreintes avec et sans oisiveté. 65
5.3 Exemple de non résistance aux augmentations de périodes. 66
5.4 Exemple de scénario synchrone et asynchrone. 69
5.5 Comparaison de l'ordonnançabilité d'algorithmes FTP : algorithme global, algorithme à migrations restreintes et r-SP_wl. 69
5.6 Comparaison des bornes basées sur le LOAD de l'algorithme global et de r-SP_wl. 72

6.1 Diagramme de représentation des algorithmes de génération de tâches. . 78
6.2 Diagramme de représentation des algorithmes de calcul de marge. . . . 79
6.3 Diagramme de représentation des algorithmes de partitionnement. . . . 79
6.4 Diagramme de représentation des algorithmes d'ordonnancement. . . . 81

A.1 Exemple d'ordonnancement avec MPCP. 104

A.2 Exemple d'ordonnancement avec MSRP 105
A.3 Exemple d'ordonnancement avec FMLP (ressources longues). 106
A.4 Exemple d'ordonnancement avec FMLP (ressources courtes). 107

LISTE DES TABLEAUX

1.1 Classes d'algorithmes d'ordonnancement temps réel. 12
1.2 Classes d'algorithmes d'ordonnancement temps réel multiprocesseur. . 14

2.1 Notations employées dans le chapitre 2. 20

3.1 Notations employées dans le chapitre 3. 30
3.2 Algorithmes de partitionnement 37

4.1 Notations employées dans le chapitre 4. 47

5.1 Notations employées dans le chapitre 5. 58
5.2 Paramètres temporels des instances de l'exemple de la figure 5.1. 60

A.1 Notations pour MPCP. 103
A.2 Notations pour MSRP. 105
A.3 Notations pour FMLP. 106

CHAPITRE 1

INTRODUCTION GÉNÉRALE

1.1 Systèmes temps réel

Cette thèse s'inscrit dans la thématique de l'informatique « temps réel ». Le terme temps réel peut revêtir plusieurs sens. Il est souvent associé à l'absence de latence ou d'attente. On fera référence à une « animation temps réel » pour parler d'une animation qui peut être visionnée en même temps qu'elle est produite (par opposition à une animation pré-calculée). La production de telles animations est rendue possible par la puissance de calcul grandissante des ordinateurs actuels. Ce terme de temps réel se prête aussi très bien à des néologismes tels que l'« information temps réel », qui ne fait finalement référence qu'à de l'information distribuée avec des délais brefs, grâce notamment aux moyens de communication modernes.

Pourtant, l'informatique temps réel n'est pas une thématique récente issue des nouvelles technologies. Au contraire, le besoin de déterminisme dans les systèmes informatiques complexes s'est très vite fait ressentir, notamment lorsqu'il est devenu possible d'utiliser l'outil informatique pour contrôler des *systèmes critiques**.

Définition 1.1 (Système critique)**.** *Un système critique est un système dont une défaillance peut avoir des conséquences dramatiques, telles que des morts ou des blessés graves, des dommages matériels importants, ou encore des conséquences graves pour l'environnement.*

L'utilisation de tels systèmes pouvant comporter des risques, il est donc nécessaire d'apporter un soin particulier au développement des programmes qui les contrôlent. En informatique temps réel, ces programmes sont soumis à des contraintes temporelles.

Définition 1.2 (Informatique temps réel [Sta88])**.** *En informatique temps réel, la validité du système ne dépend pas seulement du résultat logique du calcul, mais aussi de la date à laquelle sont produits les résultats.*

Pour se convaincre de l'utilité pratique de l'étude des *systèmes temps réel**, il convient de présenter un exemple. Un des plus pertinents est celui des systèmes embarqués dans les automobiles récentes. Il existe depuis plusieurs années des systèmes de correction électronique de trajectoire sur ces véhicules qui permettent de corriger des erreurs de conduite. Ces systèmes ont besoin pour fonctionner de connaître l'état du véhicule. Et cet état peut être obtenu en procédant à des mesures par le biais de capteurs. Mais il y a bien sûr un délai entre la prise de mesures et la transmission de l'information au système de décision. Si un capteur mesure la vitesse de rotation d'une roue, il n'est pas difficile de se convaincre que la validé de l'information fournie par ce capteur est limitée dans le temps. En effet, si la vitesse de rotation met trop de temps à arriver au calculateur, celui-ci risque de prendre une décision erronée. Il faut donc pouvoir garantir que l'acheminement, ou encore le traitement de l'information puisse être réalisé en respectant des contraintes temporelles préalablement établies.

Ce manuscrit s'inscrit dans la thématique de l'*ordonnancement** d'un ensemble de *tâches temps réel**. Il s'agit du traitement de ces tâches avec la contrainte qu'elles se terminent toutes avant une date préalablement fixée. Nous définissons avec plus de précision l'ordonnancement temps réel dans la section 1.3. Une tâche, au sens informatique du terme, peut être caractérisée par l'exécution d'une suite d'instructions sur un ordinateur.

Définition 1.3 (Tâche temps réel)**.** *Une tâche temps réel est une tâche dont l'exécution doit être terminée avant une date fixée.*

Si un système temps réel n'était composé que d'une seule tâche, il n'y aurait pas de problème d'ordonnancement. C'est pourquoi nous considérons un ensemble de tâches temps réel et nous disons qu'elles constituent une application temps réel. Cette application a besoin d'une plate-forme pour pouvoir être exécutée. Une plate-forme est constituée d'un ensemble de composants qui interagissent pour pouvoir exécuter le code de l'application. Le composant principal sur lequel se porte notre attention est le processeur.

Définition 1.4 (Système temps réel)**.** *Un* système temps réel *est un système qui est en charge de l'exécution d'un ensemble de tâches temps réel sur une plate-forme donnée.*

De plus en plus, les plates-formes se composent de plusieurs processeurs. Cette évolution technologique est principalement due au fait que la production de processeurs plus puissants avec les procédés conventionnels devient très difficile. En effet, l'augmentation de la fréquence des processeurs et la réduction de la taille de la gravure atteint des limites physiques. Les processeurs deviennent notamment trop consommateurs en énergie et difficiles à refroidir. Une solution à ces problèmes se révèle être la mise en parallèle de plusieurs processeurs. Nous parlons alors de plate-forme multiprocesseur.

Définition 1.5 (Système temps réel multiprocesseur)**.** *Un* système temps réel multiprocesseur* *est un* système temps réel *dont la plate-forme est constituée de plusieurs processeurs.*

Nous présentons dans la suite de ce chapitre un ensemble de notions relatives à l'ordonnancement temps réel multiprocesseur. Dans la section 1.2, nous introduisons les modèles généralement rencontrés dans la littérature, en détaillant ceux que nous considérons dans ce manuscrit. Dans la section 1.3, nous introduisons les bases de la théorie de l'ordonnancement temps réel. Dans la section 1.4, nous faisons un rapide tour d'horizon des problématiques de l'ordonnancement temps réel multiprocesseur. Dans la section 1.5, nous présentons les raisons qui ont motivé l'écriture de cette thèse. Enfin, nous détaillons l'organisation du manuscrit dans la section 1.6.

1.2 Représentation des systèmes temps réel

Nous distinguons entre le temps réel strict et le temps réel souple. Pour un *système temps réel strict**, aucun dépassement d'échéance n'est toléré. Il s'agit là des systèmes critiques pour lesquels une défaillance peut avoir des conséquences graves. Dans un *système temps réel souple**, le dépassement de certaines contraintes temporelles est acceptable dans la mesure où la qualité du service fourni par l'application est correcte. Cette qualité de service peut par exemple s'exprimer par une borne maximale sur le dépassement des échéances [LA10]. Il s'agit des systèmes de visioconférence, ou encore de jeux en réseau. Comme nous l'avons introduit dans la section 1.1, nous nous focalisons sur l'étude des *systèmes temps réel stricts*.

Lorsqu'on étudie l'ordonnancement d'un système temps réel, il est primordial de spécifier les hypothèses qui sont faites sur ce système. Notamment, il convient de spécifier les modèles considérés car l'analyse du système dépend de ces modèles. Il existe différents modèles de tâches, ainsi que différents modèles de plates-formes multiprocesseurs. Nous présentons ici les modèles couramment rencontrés dans la littérature en précisant ceux que nous considérons dans ce manuscrit.

1.2.1 Modèles de tâches

Modèle d'activation

L'analyse d'un système temps réel implique la connaissance du modèle d'activation des tâches. Trois modèles sont couramment rencontrés dans la littérature. Le premier est celui des tâches apériodiques. Une tâche apériodique est une tâche pour laquelle la période d'activation n'est pas connue. Il est donc difficile d'analyser de manière stricte un système composé de ce type de tâches sans faire d'hypothèse sur la probabilité de leur arrivée. Les deux autres modèles d'arrivée sont respectivement ceux des *tâches*

*sporadiques** et des *tâches périodiques**. La durée séparant deux activations d'une tâche τ_i est appelée *période** dans le cas où τ_i est une tâche périodique. Dans le cas où τ_i est une tâche sporadique, la durée séparant deux activations n'est pas fixe. Néanmoins, la période minimale entre deux activations est connue, ce qui permet de pouvoir analyser ces tâches dans le pire cas. Cette *période d'inter-arrivé minimale** est, par souci d'homogénéité avec le modèle périodique, couramment appelée période. La période d'une tâche τ_i constitue un de ses paramètres temporels et nous la notons T_i. Nous portons notre attention sur l'étude des tâches sporadiques et des tâches périodiques. Le lecteur intéressé par la gestion des tâches apériodiques dans les systèmes temps réel pourra toutefois se reporter entre autres à la thèse de Masson [Mas08].

Instance

Une tâche périodique (ou sporadique) correspond à l'exécution d'un ensemble d'instructions qui se répète. Afin de distinguer les différentes occurrences d'une même tâche, nous employons le terme d'*instance** d'une tâche (aussi appelée requête). Nous notons $J_{i,k}$ la $k^{\text{ième}}$ instance de la tâche τ_i en commençant la numérotation à 1. Ainsi, la première instance de la tâche τ_1 est donc notée $J_{1,1}$. Lorsque nous faisons référence à une instance quelconque de la tâche τ_i, nous notons celle-ci J_i.

Instant d'activation

L'instant d'*activation** d'une tâche est l'instant auquel une instance de cette tâche est prête à être ordonnancée. L'instant d'activation d'une tâche correspond à un décalage par rapport à une origine des temps fixée à l'instant $t = 0$. L'instant d'activation d'une instance est appelé *release time* dans la littérature anglophone. Nous notons par conséquent cet instant d'activation $r_{i,k}$ pour l'instance $J_{i,k}$ et r_i pour une instance quelconque J_i. Nous notons O_i l'instant d'activation de la première instance de la tâche τ_i. Dans le cas d'une tâche périodique, l'instant $r_{i,k}$ est défini par $r_{i,k} = O_i + (k-1)T_i$.

Échéance

L'*échéance** d'une tâche temps réel est l'instant auquel cette tâche doit avoir terminé son exécution. C'est un paramètre commun à toutes les tâches temps réel, quel que soit le modèle considéré. Une tâche τ_i est caractérisée par son *échéance relative**, qui correspond, pour toute instance $J_{i,k}$ de τ_i, à une durée constante entre sa date d'activation $r_{i,k}$ et la date à laquelle son exécution doit être terminée. Nous notons D_i l'échéance relative d'une tâche τ_i. L'*échéance absolue** de $J_{i,k}$, que nous notons $d_{i,k}$, correspond à la durée entre l'instant 0 et l'instant où cette instance doit être terminée. Nous notons également d_i l'échéance absolue d'une instance quelconque J_i. Dans le cas d'une tâche périodique, l'instant $d_{i,k}$ est défini par $d_{i,k} = 0_i + (k-1)T_i + D_i$. Pour les tâches sporadiques ou les tâches périodiques, on différencie les modèles de tâches grâce à la

relation entre l'échéance et la période. Les trois principaux modèles référencés dans la littérature sont :

- les tâches à *échéances implicites** (ou à échéances sur requêtes) pour lesquelles $D_i = T_i$, pour toute tâche τ_i, comme étudiées dans [LL73] ;
- les tâches à *échéan-ces contraintes** pour lesquelles $D_i \leq T_i$, pour toute tâche τ_i, comme étudiées dans [Mok83] ;
- les tâches à *échéances arbitraires** pour lesquelles l'échéance D_i n'est plus contrainte par la période T_i.

Dans ce manuscrit, nous concentrons notre attention sur l'étude des ensembles de tâches à échéances contraintes. Avec ce modèle de tâche, il est possible de représenter la majeure partie des applications. De plus, les tâches à échéances arbitraires sont plus complexes à analyser. Il convient de noter que les tâches à échéances contraintes incluent les tâches à échéances implicites.

Durée d'exécution

Une tâche temps réel τ_i est caractérisée par sa durée d'exécution pire cas (*Worst Case Execution Time*) (WCET). Il s'agit de la plus longue durée pendant laquelle une instance de τ_i peut s'exécuter. En effet, l'une des motivations majeures de l'étude des systèmes temps réel est de garantir que toutes les tâches respectent leur échéance. Il est donc nécessaire de connaître quelle sera la pire durée d'exécution des tâches. Pour cela, toutes les tâches sont étudiées indépendamment. Leur WCET est estimé soit en les exécutant un grand nombre de fois (analyse dynamique), soit par étude du code (analyse statique) [CP01]. Nous notons C_i le WCET de la tâche τ_i. Nous notons également $e_{i,k}$ la durée d'exécution de l'instance $J_{i,k}$ et e_i la durée d'exécution d'une instance J_i. Cette notation nous permet de distinguer deux instances avec des durées d'exécution différentes.

Notations

Dans ce manuscrit, nous considérons les ensembles de tâches périodiques et les ensembles de tâches sporadiques. Une tâche τ_i sera alors caractérisée par le quadruplet (O_i, C_i, D_i, T_i), où :

- O_i correspond à l'instant d'activation de la première instance de la tâche τ_i par rapport à l'instant 0 ;
- C_i correspond au WCET de la tâche τ_i ;
- D_i correspond à l'échéance relative de la tâche τ_i ;
- T_i correspond à la période de la tâche τ_i.

De la même manière, l'instance $J_{i,k}$ de la tâche τ_i sera caractérisée par le triplet $(r_{i,k}, e_{i,k}, d_{i,k})$, où :

- $r_{i,k}$ correspond à l'instant d'activation de l'instance $J_{i,k}$,

– $e_{i,k}$ correspond au coût d'exécution de l'instance $J_{i,k}$,
– $d_{i,k}$ correspond à l'échéance absolue de l'instance $J_{i,k}$.

Lorsque nous faisons référence à une instance quelconque de la tâche τ_i, nous employons la notation J_i caractérisée par le triplet (r_i, e_i, d_i).

Définition 1.6 (Utilisation d'une tâche)**.** *L'utilisation U_i d'une tâche τ_i représente la fraction du temps processeur nécessaire pour pouvoir exécuter τ_i dans le pire cas et est définie par :*

$$U_i = \frac{C_i}{T_i}$$

Par exemple, une tâche caractérisée par $(O_i = 0, C_i = 2, D_i = 6, T_i = 10)$ a une utilisation de 0.2. En effet, elle nécessite dans le pire cas 20% de la capacité de calcul d'un processeur pour pouvoir être exécutée.

Définition 1.7 (Utilisation d'un ensemble de tâches)**.** *L'utilisation $U(\tau)$ d'un ensemble de tâches τ représente la capacité de calcul nécessaire pour pouvoir exécuter τ dans le pire cas et est définie par :*

$$U(\tau) = \sum_{i=1}^{n} U_i$$

Indépendance

Une hypothèse souvent rencontrée dans la littérature est celle de l'indépendance des tâches. Cela signifie que les seules ressources pour lesquelles une tâche du système est en concurrence sont l'ensemble des processeurs en charge de l'exécuter. Pourtant, lorsque l'on considère une application réelle, il est fort probable que les différentes tâches qui la composent ne soient pas indépendantes. En effet, il peut exister des contraintes de précédences entre des tâches, c'est-à-dire qu'une tâche requiert qu'une autre ait terminé son exécution avant de commencer la sienne. L'application peut aussi nécessiter que des ressources (autres que les processeurs) soient partagées entre les différentes tâches. Dans ce manuscrit, nous considérons principalement des modèles de tâches indépendantes. Le problème du partage de ressource est toutefois abordé dans le chapitre 4.

1.2.2 Temps de réponse

Définition 1.8 (Temps de réponse d'une instance)**.** *Le temps de réponse d'une instance est la durée entre la date d'activation de cette instance et sa date de terminaison.*

Une instance J_i d'une tâche τ_i respecte son échéance lorsque son *temps de réponse** est inférieur à la durée D_i (l'échéance relative de τ_i).

Définition 1.9 (Pire temps de réponse d'une tâche)**.** *Le pire temps de réponse d'une tâche est le temps de réponse de l'instance de cette tâche qui a le temps de réponse le plus long.*

Une tâche τ_i ne dépasse aucune de ses échéances lorsque son pire temps de réponse est inférieur à son échéance relative D_i.

1.2.3 Modèles de processeurs

Concernant les plates-formes multiprocesseurs, il est courant de rencontrer dans l'état de l'art trois modèles d'architecture :

- les systèmes uniformes. Sur un tel système, chaque processeur π_j est caractérisé par sa capacité de calcul s_j. Ainsi, toute instance exécutée sur un processeur π_j pendant une durée t réalise $s_j \times t$ unités d'exécution. Un exemple de ce type de système est représenté par une plate-forme avec des processeurs identiques dont la fréquence de chaque cœur peut être fixée indépendamment ;
- les systèmes homogènes (ou identiques). Ils sont caractérisés par $\forall\ j, k,\ 1 \leq j, k \leq m,\ s_j = s_k$. Un exemple de ce type de systèmes est représenté par la plupart des processeurs multi-cœurs grand public ;
- les systèmes hétérogènes. Pour ces systèmes, un taux $\rho_{i,j}$ est associé à chaque paire formée de l'instance J_i et du processeur π_j. Ainsi, une instance J_i exécutée sur un processeur π_j pendant une durée t réalise $\rho_{i,j} \times t$ unités d'exécution. Un exemple de ce type de systèmes est le processeur Cell d'IBM (notamment utilisé dans la PlayStation® 3 de Sony). Ces processeurs sont constitués d'un cœur principal et de 8 cœurs spécifiques.

Ces trois modèles prennent uniquement en considération la capacité de calcul des processeurs. Il est également possible de distinguer différents types d'accès à la mémoire. La plupart des processeurs multi-cœurs grand public sont à accès mémoire uniforme (*Uniform Memory Access*) (UMA). Cela signifie que chaque cœur accède à la mémoire centrale via un unique canal de communication partagé. Mais il existe aussi des plates-formes à accès mémoire non uniforme (*Non-Uniform Memory Access*) (NUMA). Par exemple, une plate-forme peut être constituée de plusieurs processeurs Opteron d'AMD montés en parallèle. Chaque processeur dispose de sa propre mémoire en accès rapide, mais peut aussi avoir accès à la mémoire des autres processeurs en accès plus lent. Les deux distinctions précédemment évoquées portent sur l'aspect matériel des plates-formes. Il existe une distinction logicielle entre le multitraitement symétrique (*Symmetric multiprocessing*) (SMP) et le multitraitement asymétrique (*Asymmetric multiprocessing*) (ASMP). En SMP, chaque processeur est en mesure d'exécuter n'importe quelle tâche, tandis qu'en ASMP, un ensemble de tâches ne pourra être attribué qu'à un ou plusieurs processeurs. Dans ce manuscrit, nous considérons le modèle le plus simple, à savoir une plate-forme multiprocesseur homogène, UMA et SMP.

1.3 Ordonnancement temps réel

Pour un système de tâches, l'ordonnancement consiste à décider dans quel ordre exécuter les tâches. Un ordonnancement temps réel strict a le même objectif mais avec la contrainte supplémentaire qu'aucune instance d'aucune tâche ne rate son échéance. Une des motivations de la théorie de l'ordonnancement, et par extension de l'ordonnancement temps réel, est de concevoir des algorithmes d'ordonnancement qui minimisent les temps de réponse des tâches. Dans le domaine du temps réel, en plus de décrire un algorithme d'ordonnancement, il est nécessaire de proposer un algorithme qui garantit, pour cet ordonnancement, que les échéances soient respectées.

1.3.1 Ordonnancement hors-ligne / en ligne

L'ordonnancement des tâches temps réel est à la charge de l'ordonnanceur, qui est une des briques logicielles qui composent le système d'exploitation temps réel. L'ordonnanceur peut être vu comme une tâche spécifique chargée de décider dans quel ordre exécuter les autres tâches. On distingue les *ordonnancements hors-ligne** des *ordonnancements en ligne**. Pour un ordonnancement hors-ligne, les décisions d'ordonnancement sont connues avant le début de l'ordonnancement. L'ordonnanceur n'a donc pas à calculer ses décisions d'ordonnancement, mais juste à appliquer une politique déjà établie. Au contraire, pour un ordonnancement en ligne, l'ordonnanceur prend ses décisions durant l'ordonnancement en fonction de l'état du système.

1.3.2 Ordonnancement préemptif / non préemptif

Lorsque plusieurs tâches doivent être ordonnancées de manière concurrente, deux approches sont distinguées : les *ordonnancement non préemptifs** et *ordonnancements préemptifs**. Dans un ordonnancement non préemptif, une tâche qui a été démarrée ne peut pas être interrompue avant d'avoir terminé son exécution. Avec cette approche, il y a au plus un changement de contexte, c'est-à-dire que l'ordonnanceur remplace le contexte d'exécution d'une instance par celui d'une autre instance. Ces changements de contexte ont un coût qu'il faut prendre en considération durant l'analyse d'ordonnançabilité, bien que l'hypothèse que ce coût soit nul est souvent faite. Dans un ordonnancement préemptif, l'ordonnanceur peut décider d'interrompre une tâche pour en démarrer une autre. L'avantage de cette approche est de réduire les temps de réponse des tâches les plus prioritaires, de mieux utiliser les processeurs et d'obtenir des taux d'ordonnançabilité plus importants. Dans le cas d'un ordonnancement conduit par les priorités, ce type d'interruption se produit lorsqu'une instance plus prioritaire que l'instance en cours d'exécution est activée. Avec ce type d'ordonnancement, des tâches se retrouvent en situation de compétition, notamment lorsqu'elles partagent des ressources communes. Ce problème nécessite de synchroniser les ressources partagées

pour garantir leur cohérence. Dans ce manuscrit, nous nous concentrons sur l'étude des algorithmes d'ordonnancement préemptif. Le lecteur intéressé par les résultats sur l'ordonnancement non préemptif pourra se référer entre autres à [GRS96].

1.3.3 Ordonnancement oisif / non oisif

Un algorithme d'ordonnancement peut produire un *ordonnancement oisif**, c'est-à-dire qu'il peut créer un ordonnancement dans lequel un processeur est inactif alors qu'une instance d'une tâche est active. Cette propriété peut sembler contre-productive, mais elle peut représenter un intérêt dans le cas de la prédictibilité d'un algorithme d'ordonnancement comme nous le verrons dans le chapitre 5. Un *ordonnancement non oisif** ne permet pas qu'un processeur soit inactif alors qu'une instance est active. Cette propriété est appelée *work-conserving* dans la littérature anglophone.

1.3.4 Ordonnançabilité et faisabilité

Dans un système temps réel strict, la garantie du respect des échéances est obtenue par une analyse d'ordonnançabilité.

Le temps de réponse d'une instance d'une tâche est la durée entre l'instant d'activation de cette instance et l'instant de sa terminaison [JP86, AATW93].

Définition 1.10 (Ordonnançabilité d'une tâche)**.** *Une tâche est ordonnançable relativement à un algorithme d'ordonnancement si son pire temps de réponse avec cet algorithme est inférieur ou égal à son échéance.*

Le fait qu'une tâche soit ordonnançable ne garantit en rien l'ordonnançabilité des autres tâches.

Définition 1.11 (Ordonnançabilité d'un ensemble de tâches)**.** *Un ensemble de tâches temps réel est ordonnançable relativement à un algorithme d'ordonnancement si toutes les tâches qui le composent sont ordonnançables.*

Pour décider de l'ordonnançabilité d'un ensemble de tâches relativement à un algorithme d'ordonnancement, il est nécessaire de disposer d'un test d'ordonnançabilité pour cet algorithme d'ordonnancement.

Il n'est pas toujours possible, pour un algorithme donné, de pouvoir procéder à une analyse de *temps de réponse* pire cas avec une complexité en temps raisonnable. C'est pourquoi pour certaines approches d'ordonnancement, seuls des tests d'ordonnancement suffisants permettent de garantir l'ordonnancement des ensembles de tâches. L'inconvénient de ces tests est qu'ils peuvent décider qu'un ensemble de tâches n'est pas ordonnançable alors qu'il l'est effectivement.

Définition 1.12 (Test d'ordonnançabilité suffisant)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. Un test d'ordonnançabilité est suffisant relativement à $\mathcal{A}$ si tous les ensembles de tâches considérés ordonnançables par ce test sont effectivement ordonnançables.*

Pour qu'un test d'ordonnançabilité n'écarte pas des systèmes ordonnançables, il doit aussi être nécessaire.

Définition 1.13 (Test d'ordonnançabilité nécessaire)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. Un test d'ordonnançabilité est nécessaire relativement à $\mathcal{A}$ si tous les ensembles de tâches considérés non ordonnançables par ce test ne sont effectivement pas ordonnançables.*

Un test d'ordonnançabilité nécessaire trivial pour tout algorithme d'ordonnancement monoprocesseur est $\sum_{\tau_i \in \tau} U_i \leq 1$. Dans le cas multiprocesseur, ce test trivial devient $\sum_{\tau_i \in \tau} U_i \leq m$ où m est le nombre de processeurs. En effet, des processeurs ne pourront jamais exécuter un ensemble de tâches ayant besoin d'une capacité de calcul supérieure à celle que ceux-ci peuvent fournir. Mais un test nécessaire n'est pas suffisant pour pouvoir décider de l'ordonnançabilité d'un ensemble de tâche. Il doit être associé à un test d'ordonnançabilité suffisant. Réciproquement, un test suffisant doit être nécessaire pour garantir l'ordonnançabilité d'un système.

Définition 1.14 (Test d'ordonnançabilité nécessaire et suffisant)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. Un test d'ordonnançabilité est nécessaire et suffisant relativement à $\mathcal{A}$ si tous les ensembles de tâches considérés (i) ordonnançables par ce test sont effectivement ordonnançables et (ii) non ordonnançables par ce test ne sont effectivement pas ordonnançables.*

Relativement à la définition 1.11, il n'est possible de garantir le respect des échéances d'un ensemble de tâches que si l'algorithme d'ordonnancement est connu. Une notion plus générale est celle de faisabilité.

Définition 1.15 (Faisabilité d'un ensemble de tâches)**.** *Un ensemble tâches de temps réel est faisable si il existe un algorithme d'ordonnancement qui peut ordonnancer cet ensemble de telle manière que toutes les tâches qui le compose respectent leurs échéances.*

1.3.5 Priorités dans l'ordonnancement

Nous distinguons trois classes d'algorithmes d'ordonnancement. La première est référencée dans la littérature comme la classe des algorithmes à « priorités fixes » ou encore à « priorités statiques ». La dénomination exacte de cette classe devrait être la classe des algorithmes à priorités fixes au niveau des tâches (*Fixed Task Priority*) (FTP). Pour les algorithmes de cette classe, la priorité est une caractéristique de la tâche. Toutes les instances d'une tâche sont exécutées avec la priorité de cette dernière. Les algorithmes les plus connus de cette classe sont les algorithmes *Rate-Monotonic (RM)*

[LL73], *Deadline-Monotonic (DM)* [LW82] et l'algorithme d'attribution des priorités optimal (*Optimal Priority Assignment*) (OPA) [Aud01, DB95]. Nous présentons ces algorithmes dans la section 1.3.6.

La deuxième classe d'algorithmes est souvent référencée dans la littérature comme la classe des algorithmes à « priorités dynamiques ». La dénomination exacte de cette classe devrait être la classe des algorithmes à priorités fixes au niveau des instances (*Fixed Job Priority*) (FJP). En effet, les algorithmes de cette classe attribuent des priorités fixes aux instances qui ne changent pas pendant l'exécution de ces instances. Ainsi, les priorités de deux instances d'une même tâche peuvent être différentes. L'algorithme le plus représentatif de cette classe est l'algorithme *Earliest Deadline First (EDF)* [LL73]. Il attribue la priorité la plus haute à l'instance ayant l'échéance absolue la plus courte.

La troisième classe d'algorithmes est souvent, à juste titre, référencée comme la classe des algorithmes à priorités dynamiques (*Dynamic Priority*) (DP), même si l'amalgame est souvent fait avec les algorithmes FJP. Les algorithmes de cette classe font évoluer la priorité à l'intérieur des instances en fonction de l'état du système. La priorité peut changer pendant l'exécution de l'instance. L'algorithme le plus représentatif de cette classe est *Least Laxity First (LLF)* [Leu89, DM89]. Il attribue à l'instant t la priorité la plus forte à l'instance dont la durée entre son instant de terminaison et son échéance est la plus courte.

La différence entre les classes FJP et DP est le moment où la priorité d'une instance est recalculée, c'est-à-dire le moment où l'ordonnanceur est invoqué. Dans le cas de la classe FJP, la priorité de l'instance est recalculée au moment de son activation. Dans le cas de la classe DP, elle est recalculée à l'activation de toute instance.

Définition 1.16 (Algorithme d'ordonnancement optimal)**.** *Un algorithme d'ordonnancement est optimal relativement à un modèle de tâche si il peut ordonnancer tous les ensembles de tâches faisables relativement à ce modèle.*

Pour les plates-formes monoprocesseurs, l'algorithme d'ordonnancement EDF a été prouvé optimal pour des ensembles de tâches périodiques à échéances implicites par Liu et Layland [LL73]. Dertouzos a étendu ce résultat aux ensembles de tâches périodiques à échéances contraintes ou à échéances arbitraires [Der74]. Un ensemble de tâches périodiques est donc faisable si il est ordonnançable avec EDF. Décider de la faisabilité d'un tel ensemble de tâches revient ainsi à tester son ordonnançabilité avec EDF. Le test d'ordonnançabilité nécessaire et suffisant d'EDF pour des tâches à échéances implicites est que l'utilisation de l'ensemble de tâches τ soit inférieure ou égale à la capacité de calcul du processeur. Cette condition s'exprime par la relation $U(\tau) \leq 1$. Un ensemble de tâches périodiques à échéances implicites respectant cette condition est donc ordonnançable avec EDF. EDF étant optimal pour ce modèle de tâche, cet ensemble est donc faisable. De la même manière, l'algorithme d'ordonnancement LLF a été prouvé optimal pour des ensembles de tâches périodiques à échéances contraintes [Mok83].

Priorités fixes au niveau des tâches	Priorités fixes au niveau des instances	Priorités dynamiques
RM [LL73], DM [LW82], OPA [Aud01]	EDF [LL73]	LLF [Mok83]

TABLE 1.1 – Classes d'algorithmes d'ordonnancement temps réel.

Dans cette thèse, nous focalisons notre attention sur la classe des algorithmes FTP. En effet, les ordonnanceurs à priorités fixes (au niveau des instances) ont l'avantage d'être simple à implanter. Ce qui implique qu'ils sont largement supportés dans les systèmes d'exploitation temps réel et que des interfaces sont disponibles pour pouvoir les exploiter (par exemple POSIX ou AUTOSAR). Afin d'éviter toute ambiguïté, nous désignons cette classe par l'acronyme FTP dans suite de ce manuscrit.

1.3.6 Attribution des priorités

Les algorithmes d'ordonnancement FTP fixent une priorité à chaque tâche et celle-ci n'évolue plus durant l'ordonnancement. On parle alors d'algorithme d'attribution des priorités. L'algorithme RM fixe la priorité la plus haute à la tâche ayant la période la plus petite, tandis que l'algorithme DM fixe la priorité la plus haute à la tâche ayant l'échéance la plus petite.

Définition 1.17 (Algorithme d'attribution des priorités optimal)**.** *Un algorithme d'attribution des priorités est optimal relativement à un modèle de tâche si il peut trouver une attribution faisable des priorités pour un ensemble de tâches ordonnançables en FTP.*

L'algorithme RM est optimal dans le cas des tâches à échéances implicites. Mais il faut être vigilant avec ce résultat. Cela ne veut pas dire qu'un ensemble de tâches faisable pourra forcément être ordonnancé avec RM. Cela signifie que si il existe une attribution des priorités telle que l'ensemble de tâches soit ordonnançable en FTP, alors RM peut ordonnancer cet ensemble de tâches. De la même manière, DM est un algorithme d'attribution des priorités optimal pour les tâches à échéances contraintes. OPA, comme son nom l'indique, est aussi un algorithme d'attribution des priorités optimal. Il présente l'avantage de ne pas dépendre d'un modèle de tâche particulier.

1.4 Ordonnancement temps réel multiprocesseur

L'ordonnancement temps réel tel qu'il est décrit depuis le début de ce chapitre concerne principalement les plates-formes monoprocesseurs. Dans cette thèse, nous nous intéressons aux plates-formes composées de plusieurs processeurs. Ce type de plate-forme est désigné par le terme de plate-forme multiprocesseur. Depuis le début des années 80, le constat qu'il serait impossible d'augmenter indéfiniment la fréquence des processeur a motivé l'intérêt pour les architectures à plusieurs processeurs.

Le problème de la consommation énergétique et de la dissipation thermique a renforcé cette intérêt. C'est pourquoi de plus en plus de plates-formes multiprocesseurs (et notamment multi-cœurs) sont proposées sur le marché. Bien que la motivation en terme d'ordonnancement reste la même, à savoir garantir le respect des échéances pour un ensemble de tâches, le nombre de possibilités d'ordonnancement augmente.

En ordonnancement temps réel monoprocesseur, nous pouvons considérer que l'ordonnanceur n'a qu'un seul degré de liberté, à savoir une liberté temporelle. En effet, le problème d'ordonnancement consiste à décider quand exécuter une tâche sur ce processeur. Ce problème est en général résolu, comme nous l'avons présenté dans la section 1.3, en appliquant des priorités, qu'elles soient fixes ou dynamiques. Dans le cas de l'ordonnancement multiprocesseur, l'ordonnanceur a un degré de liberté supplémentaire, à savoir une liberté spatiale. En plus de décider quand exécuter une tâche, l'ordonnanceur doit maintenant décider sur quel processeur l'exécuter. Ainsi, l'ordonnanceur peut décider de faire migrer une tâche ou une instance de tâche d'un processeur vers un autre.

Dans l'état de l'art, il est fréquemment énoncé qu'en ordonnancement temps réel multiprocesseur, deux principales approches sont distinguées par leur aptitude à tolérer ou non les *migrations** entre processeurs. La première est l'ordonnancement par partitionnement, pour laquelle les migrations ne sont pas autorisées. Chaque processeur dispose alors de son propre ordonnanceur pour ordonnancer l'ensemble de tâches qui lui a été alloué. La seconde est l'ordonnancement global, pour laquelle les migrations sont autorisées. Avec cette approche, un ordonnanceur unique ordonnance toutes les tâches du système. Un des arguments qui peut justifier que la communauté porte un intérêt à ces deux approches est qu'elles sont incomparables. En d'autres termes, il existe des ensembles de tâches qui sont ordonnançables avec un algorithme d'ordonnancement par partitionnement et ne le sont pas avec un algorithme d'ordonnancement global, et réciproquement (il existe des ensembles de tâches qui sont ordonnançables avec un algorithme d'ordonnancement global et ne le sont pas avec un algorithme d'ordonnancement par partitionnement).

Ces approches sont relativement différentes dans le sens où elles ne soulèvent pas les mêmes problèmes. L'étude des ordonnancements globaux soulève le problème de l'ordonnançabilité des ensembles de tâches. Par contre, la problématique majeure de l'ordonnancement par partitionnement est de trouver une partition de l'ensemble des tâches sur les différents processeurs composant la plate-forme d'exécution. Une fois la partition trouvée, l'ordonnancement de chaque sous-ensemble de tâches est indépendant. L'ordonnancement de chaque sous-ensemble de tâches peut donc être réalisé par un algorithme d'ordonnancement monoprocesseur. Sans rentrer dans les détails algorithmiques, trouver une partition d'un ensemble de tâches sur plusieurs processeurs est un problème *NP-difficile*. Cela signifie qu'il n'existe pas d'algorithme de partitionnement qui garantisse de trouver une solution optimale en temps polynomial, à moins

que $P = NP$. La conjecture que l'ensemble des problèmes solubles en temps polynomial par une machine de Turing déterministe (P) ne soit pas égal à l'ensemble des problèmes solubles en temps polynomial par une machine de Turing non déterministe (NP) sort du cadre de ce manuscrit et ne sera donc pas détaillée.

L'ordonnancement par partitionnement consiste à allouer une tâche sur un processeur. Chaque instance de cette tâche est alors exécutée sur ce processeur. Par analogie avec les classes d'algorithmes d'attribution des priorités, nous pouvons qualifier cette approche d'ordonnancement multiprocesseur d'ordonnancement à affectation fixe des tâches aux processeurs (*Fixed Task Processor*) (FTΠ). De la même manière, un ordonnancement global peut être qualifié d'ordonnancement à affectation dynamique des instances aux processeurs (*Dynamic Processor*) (DΠ). En effet, toute instance d'une tâche peut migrer plusieurs fois d'un processeur à un autre durant son exécution. À l'instar des algorithmes d'attribution des priorités, il apparaît logique de considérer l'approche d'ordonnancement multiprocesseur intermédiaire que nous pouvons qualifier d'ordonnancement à affectation fixe des instances aux processeurs (*Fixed Job Processor*) (FJΠ). Cette dernière approche, peu étudiée comparativement aux deux premières, est désignée dans la littérature comme ordonnancement à migrations restreintes. Elle présente l'intérêt de borner le nombre de migrations à au plus une par instance de tâche. Les migrations inter-processeurs ayant à l'heure actuelle un coût prohibitif, il est nécessaire de minimiser le nombre de ces migrations.

	Priorités fixes au niveau des tâches FTP	Priorités fixes au niveau des instances FJP	Priorités dynamiques DP
Processeurs fixes au niveau des tâches FTΠ	RMNF & RMFF [DL78], RMBF [OS93], RMST & RMGT [BLOS95], RM-FFDU [OS95], RM-DU-NFS [AJ02]	EDF-NF & EDF-FF [LGDG00], EDF-BF [LDG04]	[SWPT03]
Processeurs fixes au niveau des instances FJΠ	[Fis07]	r-EDF [BC03]	FLLF [YSKA+08]
Processeurs dynamiques DΠ	RM-US[m/(3m-2)] [ABJ01], Global-DM (BAK) [Bak03], Global-FP (RTA) [BC07]	GFB [GFB03], BAK, [Bak03], BAR [Bar07], RTA [BC07], LOAD [BB09a], BCL [BCL09], FF-DBF [BBMSS10]	Pfair [BCPV96], ERfair [AS00], BF [ZMM03], LLREF [CRJ06], DP-Fair [LFPB10]

TABLE 1.2 – Classes d'algorithmes d'ordonnancement temps réel multiprocesseur.

Il est possible de fixer une certaine analogie entre les classes d'algorithmes d'attri-

bution des priorités et les degrés de migrations autorisées. C'est pourquoi Carpenter *et al.* ont proposé une table similaire à la table 1.2 [CFH+04]. Ainsi, on peut faire référence à une classe d'algorithme par (X-Y) où X est la classe d'algorithme d'attribution des priorités et Y le degré de migrations autorisées. Les références aux travaux que nous citons dans cette table ne sont pas exhaustives, mais donnent une tendance quant à l'intérêt porté à ces différentes approches d'ordonnancement. L'ordonnancement par partitionnement (FTΠ) est très souvent mis en opposition à l'ordonnancement global (DΠ), de même que l'ordonnancement à priorités fixes (FTP) est mis en opposition à l'ordonnancement EDF (FJP). C'est pourquoi de nombreux travaux se focalisent sur les approches (FTP-FTΠ) et (FJP-FTΠ) d'une part, et (FTP-DΠ) et (FJP-DΠ) d'autre part. Depuis une quinzaine d'années, nous constatons aussi un engouement pour l'approche (DP-DΠ), qui s'explique par le fait que l'utilisation des priorités dynamiques couplée à l'absence de restriction sur les migrations permettent de concevoir des algorithmes optimaux.

Bien que la table 1.2 organise de manière claire les principales classes d'algorithmes d'ordonnancement multiprocesseur, elle ne les représente pas toutes. En effet, il existe des approches hybrides d'ordonnancement qui exploitent les avantages de plusieurs classes d'algorithmes. Ainsi les algorithmes *Earliest Deadline until Zero Laxity (EDZL)* [Lee94, WCL+07] et *Earliest Deadline until Critical Laxity (EDCL)* [KY11] exploitent les mécanismes des classes FJP et DP. De même, l'algorithme *Fixed Priority until Zero Laxity (FPZL)* [DB11] exploite les mécanismes des classes FTP et DP. Ces algorithmes sont hybrides relativement à leurs classes d'attribution des priorités. Il en va de même pour les classes de degré de migrations autorisées. Ainsi, il est fait référence à l'approche semi-partitionnée qui pour un ordonnancement par partitionnement autorise un sous-ensemble de tâches à migrer. Les algorithmes semi-partitionnés ont beaucoup été étudiés dans le cas FTΠ / DΠ. Nous pouvons citer les algorithme EKG [AT06], EDF-SS(DMIN /δ) [ABB08], RMDP [KY08], DM-DP [KY09], PDMS_HPTS_DS [LdNR09], EDF-WM [KYI09] et SPA2 [GSYY10] qui sont tous des algorithmes semi-partitionnés de cette classe hybride de migrations autorisées. Il existe également un algorithme pour le cas FTΠ / FJΠ [DMYGR10]. La plupart de ces algorithmes ont été implantés et comparés par George, Courbin et Sorel, qui ont également proposé l'algorithme EDF-RRJM [GCS11].

1.5 Motivations

Dans cette thèse, en plus de nous intéresser à l'ordonnançabilité des systèmes, nous focalisons notre attention sur la sûreté temporelle. Nous considérons la sûreté temporelle d'un système comme une propriété de tolérance aux variations des paramètres temporels des tâches. Nous nous intéressons à deux notions permettant d'accroître la sûreté dans les systèmes temps réel. Dans un premier temps, il s'agit de la robustesse

temporelle qui permet de donner des garanties quant aux variations tolérables des paramètres temporels lorsque ceux-ci rendent le système plus difficilement ordonnançable. Par la suite, nous considérons la notion de viabilité d'un ordonnancement. Cette notion consiste en la propriété qu'a un algorithme de pouvoir ordonnancer un ensemble de tâches alors même que les paramètres temporels peuvent varier de manière à rendre l'ensemble de tâches moins contraint. Dans le cas de temps d'exécution inférieurs au WCET, on parlera de prédictibilité. Cette notion de viabilité semble contradictoire et pourtant, certains algorithmes d'ordonnancement sont sujets à ce genre d'anomalies. Nous détaillons ce concept de sûreté temporelle plus en détail dans le chapitre 2 qui lui est consacré entièrement.

1.6 Organisation

Le reste de ce manuscrit est organisé de la manière suivante. Dans le chapitre 2, nous définissons le concept de sûreté temporelle qui a motivé la rédaction de cette thèse. Nous abordons différentes approches permettant d'apporter cette sûreté aux systèmes temps réel multiprocesseur. Dans le chapitre 3, nous focalisons notre attention sur l'approche d'ordonnancement par partitionnement et nous nous concentrons sur un modèle de tâches indépendantes. Il s'agit là d'un ordonnancement hors-ligne car l'algorithme que nous proposons ne fait que décider de l'allocation des tâches sur les processeurs. Dans le chapitre 4, nous étendons cette étude au cas du modèle des tâches partageant des ressources. L'approche d'ordonnancement considérée est toujours une approche d'ordonnancement par partitionnement. Dans le chapitre 5, nous nous intéressons cette fois à une approche d'ordonnancement en ligne. Contrairement aux ordonnancements présentés dans les deux chapitres précédents, le choix du processeur pour l'allocation d'une instance revient à l'ordonnanceur à l'instant de l'activation de celle-ci. L'algorithme d'ordonnancement que nous présentons dans ce chapitre appartient à la classe des algorithmes FTII. Dans le chapitre 6, nous présentons l'outil de simulation que nous avons développé. Enfin, dans le chapitre 7, nous faisons la synthèse des travaux présentés dans cette thèse et donnons nos perspectives de recherche.

CHAPITRE 2

SÛRETÉ TEMPORELLE

2.1 Introduction

Nous nous intéressons à la sûreté temporelle dans les systèmes temps réel multiprocesseurs. Nous devons donc définir précisément en quoi consiste cette sûreté temporelle. Tout d'abord, nous rappelons que les tâches temps réel sont des tâches contraintes par leurs échéances temporelles et que la validité du résultat calculé par une tâche dépend du respect de cette échéance. Pour pouvoir analyser l'ordonnancement d'un ensemble de tâches, il est nécessaire de connaître certains paramètres. Les deux principaux paramètres connus sont le WCET et la période. Nous définissons alors la sûreté temporelle de la manière suivante :

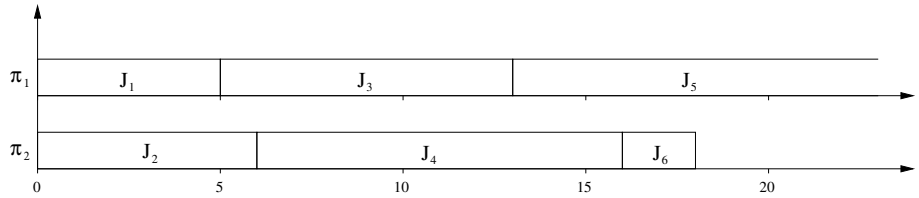
Définition 2.1 (Sûreté temporelle)**.** *Soit un algorithme d'ordonnancement $\mathcal{A}$ respectivement à un modèle de tâche donnée. La sûreté temporelle associée à un intervalle de variations $\mathcal{I}$ est la propriété pour $\mathcal{A}$ de pouvoir garantir le respect des échéances lorsque les paramètres temporels des tâches subissent des variations comprises dans $\mathcal{I}$.*

Une des propriétés de la sûreté temporelle est la capacité qu'a un algorithme à tolérer des variations négatives sur les paramètres temporels des tâches. Nous parlons dans ce cas de robustesse temporelle.

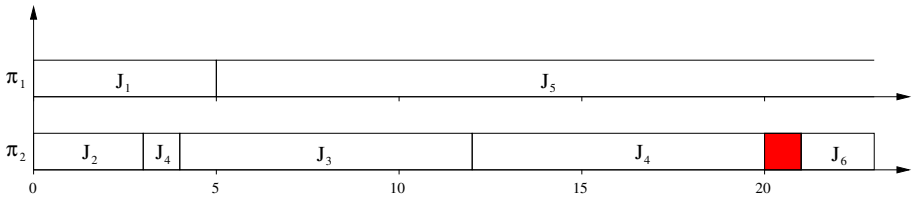
Définition 2.2 (Variation négative)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. Soit τ un ensemble de tâches ordonnançable avec $\mathcal{A}$. Une variation négative sur un paramètre temporel d'une tâche de τ est une variation qui rend τ « plus difficilement ordonnançable » avec $\mathcal{A}$. Une variation négative pour une tâche τ_i peut donc être l'augmentation de son WCET C_i ou une diminution de sa période T_i.*

Pour illustrer ce que nous entendons par « plus difficilement ordonnançable », considérons les deux tâches suivantes ordonnancées en FTP :

- τ_1 $(C_1 = 2, T_1 = 4)$ est la plus prioritaire,



(a) Ensemble d'instances ordonnançable



(b) Ensemble d'instances non ordonnançable

FIGURE 2.1 – Algorithme ne supportant pas les variations négatives sur les paramètres.

– τ_2 $(C_2 = 4, T_2 = 8)$ est la moins prioritaire.

L'ensemble de tâches $\tau = \{\tau_1, \tau_2\}$ est ordonnançable. Si nous considérons maintenant l'ensemble de tâches $\tau' = \{\tau'_1, \tau_2\}$ où τ'_1 est définie par $(C'_1 = 3, T'_1 = 4)$, celui-ci ne l'est plus. Nous pouvons dire que l'augmentation du WCET d'une tâche a rendu l'ordonnancement plus difficile. Maintenant, nous considérons l'ensemble de tâches $\tau'' = \{\tau''_1, \tau_2\}$ où τ''_1 est définie par $(C''_1 = 1, T''_1 = 4)$. Comme τ est ordonnançable, nous pouvons ajouter 1 au WCET C''_1 de τ''_1 et τ'' reste ordonnançable (puisque $\tau'' = \tau$). Nous avons ainsi pu appliquer une variation négative sur le WCET d'une tâche de τ'' de manière à ce qu'elle reste ordonnançable.

Une autre des propriétés définissant la sûreté temporelle est la capacité qu'a un algorithme à tolérer des variations positives sur les paramètres temporels des tâches.

Définition 2.3 (Variation positive)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. Soit τ un ensemble de tâches ordonnançable avec $\mathcal{A}$. Une variation positive sur un paramètre temporel d'une tâche de τ est une variation qui rend τ plus facilement ordonnançable avec $\mathcal{A}$. Une variation positive pour une tâche τ_i peut donc être la réduction de son WCET C_i, une augmentation de sa période T_i ou une augmentation de son échéance relative D_i.*

Contrairement aux variations négatives des paramètres temporels qui peuvent évidemment poser des problèmes d'ordonnancement, il n'est pas intuitif que d'appliquer des variations positives sur les paramètres temporels puisse en poser aussi. Mais comme nous pouvons le remarquer dans l'exemple proposé par Ha et Liu [HL94] et représenté dans la figure 2.1, il est possible qu'un algorithme puisse ordonnancer un ensemble d'instances mais qu'il ne puisse pas ordonnancer le même ensemble si l'une d'entre elles avait une durée d'exécution inférieure au WCET de la tâche associée. En effet, nous représentons dans la figure 2.1(a) un ensemble d'instances ordonnançable avec un algorithme $\mathcal{A}$ de la classe d'ordonnancement (FTP-FJΠ) sur une plate-forme composée de 2 processeurs. Ces instances sont indexées dans l'ordre décroissant de

leur priorité. J_1 est l'instance la plus prioritaire et J_6 la moins prioritaire. Ces instances sont caractérisées de la manière suivante : J_1 $(r_1 = 0, e_1 = 5, d_1 = 10)$, J_2 $(r_2 = 0, e_2 = 6, d_2 = 10)$, J_3 $(r_3 = 4, e_3 = 8, d_3 = 15)$, J_4 $(r_4 = 0, e_4 = 10, d_4 = 20)$, J_5 $(r_5 = 5, e_5 = 100, d_5 = 200)$ et J_6 $(r_6 = 7, e_6 = 2, d_6 = 25)$. Nous représentons dans la figure 2.1(b) ce même ensemble d'instances. Cette fois, l'instance J_2 a une durée d'exécution inférieure et est caractérisée par $(r_2 = 0, e_2 = 3, d_2 = 10)$. L'ensemble d'instances n'est plus ordonnançable. Nous détaillons cet exemple dans le chapitre 5. Ce phénomène est appelé « anomalie d'ordonnancement ». Ainsi, bien que nous puissions dire qu'un algorithme supporte des variations positives, nous pouvons également dire que cet algorithme résiste aux anomalies d'ordonnancement.

L'échéance absolue d'une tâche n'est pas seulement un paramètre de cette tâche, il s'agit également d'une contrainte temporelle. Néanmoins, il est intéressant de noter qu'un algorithme d'ordonnancement peut ne pas supporter l'augmentation d'échéance d'une ou plusieurs tâches. Nous précisons la notion de résistance aux anomalies dans la section 2.4.

Dans la section 2.2, nous définissons la terminologie et les notations employées dans ce chapitre. Dans la section 2.3, nous présentons les solutions algorithmiques permettant de calculer les variations négatives qui peuvent être tolérées par un algorithme d'ordonnancement FTP. Dans la section 2.4, nous nous intéressons cette fois aux variations positives et nous introduisons la notion de viabilité d'un ordonnancement. Enfin, dans la section 2.5, nous résumons les notions qui ont été présentées dans ce chapitre.

2.2 Terminologie et notations

Nous notons J_i^k un ensemble de k instances de la tâche τ_i $J_i^k = \{J_{i,1}, \ldots, J_{i,k}\}$. Nous notons également $J_i(p)$ l'ensemble des p instances les plus prioritaires $J_i(p) = \{J_{i,1}, \ldots, J_{i,p}\}$ avec $p < k$ et $p, k \in \mathbb{N}$. Pour rappel, nous notons J_i une instance de la tâche τ_i. Cette instance est caractérisée par le triplet (r_i, e_i, d_i) où r_i est l'instant de son activation, e_i est sa durée d'exécution et d_i son échéance absolue. Nous notons J_i^+ une instance de la tâche τ_i caractérisée par (r_i, e_i^+, d_i) où e_i^+ est la durée d'exécution le plus grand parmi toutes les instances de la tâche τ_i. De la même manière, nous notons J_i^- une instance de la tâche τ_i caractérisée par (r_i, e_i^-, d_i) où e_i^- est la durée d'exécution le plus petit. Ainsi, nous notons $J_i^+(p)$ l'ensemble $\{J_{i,1}^+, \ldots, J_{i,p}^+\}$ et $J_i^-(p)$ l'ensemble $\{J_{i,1}^-, \ldots, J_{i,p}^-\}$. Nous notons l'instant où démarre l'instance la moins prioritaire de J_i^k par $S(J_i^k)$. De la même manière, nous notons l'instant où termine l'instance la moins prioritaire de J_i^k par $F(J_i^k)$. L'ensemble de ces notations est résumé dans la table 2.1.

Notation	Définition
π_j	Le processeur d'indice j
τ	Un ensemble de tâches
$\tau(\pi_j)$	L'ensemble des tâches allouées sur le processeur π_j
τ_i	La tâche d'indice i
C_i	Le WCET de τ_i
D_i	L'échéance relative de τ_i
$U(\tau)$	L'utilisation processeur de l'ensemble de tâches τ
J_i^k	Un ensemble de k instances de τ_i
J_i	Une instance de τ_i
$J_(p)$	L'ensemble des p premières instances de plus haute priorité
r_i	L'instant d'activation d'une instance J_i
e_i	La durée d'exécution d'une instance J_i ($e_i \leq D_i$)
e_i^+	La durée d'exécution maximale d'une instance J_i
e_i^-	La durée d'exécution minimale d'une instance J_i
d_i	L'échéance absolue d'une instance J_i
$J_i^+(p)$	L'ensemble des p premières instances ayant une durée d'exécution maximale
$J_i^-(p)$	L'ensemble des p premières instances ayant une durée d'exécution minimale
$S(J)$	L'instant où démarre l'instance la moins prioritaire de J
$F(J)$	L'instant où termine l'instance la moins prioritaire de J
	processeur π_j
$hp(i)$	L'ensemble des tâches plus prioritaires que τ_i
$lp(i)$	L'ensemble des tâches moins prioritaires que τ_i
W_i	Le temps de réponse pire cas de la tâche τ_i

TABLE 2.1 – Notations employées dans le chapitre 2.

2.3 Augmentation des paramètres temporels

La notion de tolérance aux variations négatives des paramètres temporels a été récemment abordée par Davis et Burns [DB07]. Ils se sont intéressés à proposer une assignation des priorités qui maximise la tolérance aux variations négatives de WCET dans le cas d'un algorithme d'ordonnancement FTP. En effet, les tâches temps réel peuvent être sujettes à des interférences additionnelles qui peuvent être dues :

- aux interruptions matérielles ;
- aux latences du système d'exploitation temps réel qui seraient mal spécifiées ou inconnues ;
- à une mauvaise estimation des WCET ;
- à du temps de calcul qui serait détourné par des contrôleurs de périphériques ;
- à des erreurs se produisant à une fréquence non prévisible, causant ainsi la réexécution d'instructions de code différentes.

Dans cette section, nous présentons des travaux dont la synthèse est faite par George dans [Geo08]. Nous rappelons ces solutions algorithmiques qui permettent de calculer les variations négatives pouvant être tolérées par un algorithme d'ordonnancement

FTP. Nous avons défini ces variations négatives comme étant l'augmentation de WCET ou la réduction de période.

2.3.1 Marge sur les paramètres temporels

Nous présentons la notion de marge sur le WCET puis nous l'étendons au domaine des périodes.

Marge sur le WCET

La marge sur le WCET en monoprocesseur, appelée *Allowance* par Bougueroua, George et Midonnet dans [BGM07], est définie de la manière suivante :

Définition 2.4 (Marge sur le WCET). *La marge sur le WCET A_i^C représente la durée d'exécution pouvant être ajoutée au WCET C_i d'une tâche τ_i sans qu'aucune tâche du système τ ne dépasse son échéance.*

Ainsi, seule la tâche τ_i dispose d'une durée A_i^C pour pouvoir continuer son exécution après avoir été exécutée pendant son WCET. Contrairement au travail présenté dans [BGM07], une seule tâche fautive est ici considérée.

Soit τ un ensemble de tâches sporadiques allouées sur le processeur π_j. La marge sur le WCET A_i^C de la tâche τ_i est la valeur maximale de marge respectant les trois inéquations :

$$U(\tau) + \frac{A_i^C}{T_i} \leq 1 \tag{2.1}$$

$$W_i^0 = C_i \tag{2.2}$$

$$W_i^{r+1} = C_i + A_i^C + \sum_{\tau_h \in hp(i)} \left\lceil \frac{W_i^r}{T_h} \right\rceil C_h \leq D_i \tag{2.3}$$

$$\forall \tau_j \in lp(i),\ W_j^0 = C_j \tag{2.4}$$

$$W_j^{r+1} = C_j + \sum_{\tau_h \in hp(j)} \left\lceil \frac{W_j^r}{T_h} \right\rceil C_h + \left\lceil \frac{W_j^r}{T_i} \right\rceil A_i^C \leq D_l \tag{2.5}$$

Dans les équations (2.3) et (2.5), W_i^r représente le temps de réponse pire cas de la tâche τ_i calculé itérativement. L'équation (2.1) est une condition nécessaire d'ordonnançabilité. Elle permet de vérifier que l'utilisation totale de l'ensemble des tâches n'excède pas la capacité de calcul du processeur. L'équation (2.3) permet de vérifier que le temps de réponse pire cas de la tâche τ_i, à laquelle est ajoutée la marge sur le WCET A_i^C à son WCET, est inférieur à son échéance relative. Enfin, l'équation (2.5) permet de vérifier que le temps de réponse pire cas de toutes les tâches moins prioritaires que τ_i est inférieur à leur échéance relative, et ce alors que le WCET de τ_i est augmenté de A_i^C. Une

recherche dichotomique permet de calculer la valeur de marge sur le WCET d'une tâche en vérifiant que les 3 équations sont respectées.

Pour pouvoir appliquer cette solution, nous devons donc fournir un intervalle de recherche pour la valeur de marge sur le WCET. La borne inférieure de cette intervalle est bien évidemment 0. Une borne supérieure évidente est donnée par la relation $D_i - C_i$. Mais dans le cas où plusieurs tâches sont allouées sur le même processeur π_j, une meilleure borne est donnée par la relation $(1 - U(\tau(\pi_j)))T_i$ où $U(\tau(\pi_j))$ est l'utilisation processeur des tâches allouées sur le processeur π_j, obtenue à partir de l'équation (2.1). Dans le cas de tâches à échéances contraintes, la relation $D_i - C_i$ peut toutefois être inférieure à la relation $(1 - U(\pi_j))T_i$. C'est pourquoi nous utilisons la relation $\min(D_i - C_i, (1 - U(\pi_j))T_i)$ (comme borne supérieure de l'intervalle de recherche de la valeur maximale de marge sur le WCET).

Marge sur la période

Nous proposons d'étendre la notion de marge sur le WCET à la période. En effet, de la même manière qu'il peut être difficile de calculer avec exactitude les valeurs du WCET, la période est un paramètre qu'il peut être dangereux de considérer comme invariable. Nous proposons un ensemble d'inéquations permettant de vérifier qu'un ensemble de tâches reste ordonnançable lorsque l'inter-arrivée d'une tâche est réduite.

Définition 2.5 (Marge sur la période)**.** *La marge sur la période A_i^f représente la valeur pouvant être ajoutée à la période T_i d'une tâche τ_i sans qu'aucune tâche du système ne dépasse son échéance.*

Soit τ un ensemble de tâches sporadiques allouées sur le processeur π_j. La valeur A_i^f est une valeur de marge sur la période valide de la tâche τ_i ($\tau_i \in \tau$) si les équations suivantes sont respectées :

$$U(\tau - \{\tau_i\}) + \frac{C_i}{T_i - A_i^f} \leq 1 \tag{2.6}$$

$$W_i^0 = C_i \tag{2.7}$$

$$W_i^{r+1} = C_i + \sum_{\tau_h \in hp(i)} \left\lceil \frac{W_i^r}{T_h} \right\rceil C_h \leq T_i - A_i^f \tag{2.8}$$

$$\forall \tau_l \in lp(i),\ W_j^0 = C_j \tag{2.9}$$

$$W_j^{r+1} = C_j + \sum_{\tau_h \in hp(j) - \{\tau_i\}} \left\lceil \frac{W_j^r}{T_h} \right\rceil C_h + \left\lceil \frac{W_j^r}{T_i - A_i^f} \right\rceil C_i \leq D_l \tag{2.10}$$

Dans les équations (2.8) et (2.10), W_i^r représente le temps de réponse pire cas de la tâche τ_i calculé itérativement. L'équation (2.6) est une condition nécessaire d'ordonnançabilité. Elle correspond à l'équation (2.1) étendue au domaine des périodes. L'équation (2.8) permet de garantir que le temps de réponse pire cas de la tâche τ_i est inférieur à son échéance relative. Nous considérons ici le cas des échéances contraintes, d'où la relation $D_i \leq T_i - A_i^f$ qui conduit à l'équation (2.8). À l'instar de l'équation (2.5), l'équation (2.10) permet de vérifier que le temps de réponse pire cas de toutes les tâches moins prioritaires que τ_i est inférieur à leur échéance relative, et ce alors que la période de τ_i est diminuée de A_i^f.

2.3.2 Analyse de sensibilité

Les travaux de Bini, Di Natale et Buttazzo [BDNB06] sur la sensibilité abordent d'une manière sensiblement différente de la notion de marge dans le sens où elle considère des variations sur un espace. Ainsi, plutôt que de vérifier que l'augmentation du WCET d'une tâche ne compromet pas l'ordonnançabilité d'un ensemble de tâches, leur approche permet de vérifier que cet ensemble de tâches reste ordonnançable lorsqu'un facteur multiplicatif est appliqué à l'ensemble des WCET.

Sensibilité du WCET

L'analyse de sensibilité consiste à explorer un ensemble d'instants d'ordonnancement. L'ensemble des instants d'ordonnancement relativement au niveau de priorité i est noté $schedP_i$ et est définit récursivement par $schedP_i = \mathcal{P}_{i-1}(D_i)$ où $\mathcal{P}_i(t)$ est défini par :

$$\begin{cases} \mathcal{P}_0(t) = t \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t) \end{cases} \tag{2.11}$$

Le facteur maximal qui peut être appliqué aux WCET est donné par :

$$\lambda^{max} = \min_{k=1}^{n} \max_{t \in schedP_k} \frac{t - \vec{n_k} \cdot \vec{C_k}}{\vec{n_k} \cdot \vec{d_k}} \tag{2.12}$$

où $\vec{C_k}$, $\vec{n_k}$ et $\vec{d_k}$ sont des vecteurs. $\vec{C_k}$ est le vecteur des WCET, il est défini par $\vec{C_k} = (C_1, C_2, \ldots, C_n)$. $\vec{n_k}$ est défini par $\vec{n_k} = \left(\left\lceil \frac{t}{T_1} \right\rceil, \left\lceil \frac{t}{T_2} \right\rceil, \ldots, \left\lceil \frac{t}{T_{k-1}} \right\rceil, 1\right)$ et $\vec{d_k}$ est un vecteur où l'élément k correspond à la valeur de marge sur le WCET de la tâche τ_k. Par exemple, il est possible de calculer la variation maximale que nous pouvons appliquer à la fréquence d'un processeur en considérant $\vec{d_k} = \vec{C_k}$ dans l'équation 2.12. En effet, λ^{max} correspond alors au coefficient multiplicateur qu'il est possible d'appliquer à tous les WCET de l'ensemble de tâches.

À partir de l'équation 2.12, il est possible de calculer la valeur de marge sur le

WCET d'une tâche τ_i. Pour cela, il suffit de prendre $\vec{d_k} = (0, \ldots, 0, \underbrace{1}_{i^{\text{ième}}\text{élément}}, 0, \ldots, 0)$.

Ainsi, $\vec{n_k} \cdot \vec{d_k} = \left\lceil \frac{t}{T_i} \right\rceil$ et nous obtenons l'équation :

$$A_i^C = \min_{k=i}^{n} \max_{t \in schedP_k} \frac{t - \vec{n_k} \cdot \vec{C_k}}{\lceil t/T_i \rceil} \tag{2.13}$$

Le calcul de la marge sur le WCET et l'analyse de sensibilité du WCET permettent de calculer la même valeur. Avec l'analyse de sensibilité, l'ensemble des instants d'ordonnancement se calcule en temps exponentiel en le nombre de tâches avec l'équation (2.11), même si le nombre d'instants d'ordonnancement distincts reste petit, ce qui présente un inconvénient. En contrepartie, l'analyse de sensibilité permet facilement d'obtenir une marge sur le WCET applicable à toutes les tâches (un facteur d'augmentation du WCET) avec l'équation (2.12).

Sensibilité de la période

Étant donné que l'échéance relative d'une tâche doit être inférieure à son temps de réponse pire cas pour être ordonnançable, nous avons la relation sur la période T_i de τ_i suivante :

$$W_i \leq D_i \Rightarrow T_i \geq \frac{W_i}{\delta_i} \tag{2.14}$$

où δ_i est la densité de la tâche et est défini par $\frac{D_i}{T_i}$.

Bini, Di Natale et Buttazzo ont prouvé que si l'équation 2.14 n'est pas respectée, alors il existe une tâche τ_j de priorité inférieure à τ_i pour laquelle le temps de réponse W_j est un multiple entier de la période minimale T_i^{min} de τ_i. Nous obtenons donc la valeur de temps de réponse W_j^i pour τ_j suivante :

$$W_j^i(I_i) = C_j + I_i C_i + \sum_{\substack{h=1 \\ h \neq i}}^{j-1} \left\lceil \frac{W_j}{T_h} \right\rceil C_h \tag{2.15}$$

où $W_j^i(I_i)$ est le temps de réponse de la tâche τ_j en considérant que τ_i interfère exactement I_i fois sur τ_j. Nous obtenons une seconde relation sur la période T_i de τ_i :

$$T_i \leq \min_{I_i} \frac{W_j^i(I_i)}{I_i} \tag{2.16}$$

La période minimale T_i^{min} de la tâche τ_i est donnée par la relation :

$$T_i^{min} = \max \left\{ \frac{W_i}{\delta_i}, \max_{k=i+1}^{n} \min_{I_k} \frac{W_k^i(I_k)}{I_k} \right\} \tag{2.17}$$

La valeur de marge sur la période donnée par l'équation :

$$A_i^f = T_i - T_i^{min} \tag{2.18}$$

Comme pour la marge sur le WCET, le calcul de marge sur la période et l'analyse de sensibilité de la période permettent de calculer la même valeur de marge. Il n'y a pas, cette fois-ci, de différence de complexité notable entre les deux approches de calcul.

2.4 Anomalies d'ordonnancement

La sûreté temporelle relativement aux variations négatives sur les paramètres temporels (robustesse temporelle) est une propriété supplémentaire des algorithmes d'ordonnancement. Elle apporte une valeur ajoutée qui est de pouvoir tolérer, pour un intervalle de variations préalablement calculé, une mauvaise estimation des paramètres temporels ou une défaillance de la plate-forme. Par contre, la sûreté temporelle relativement aux variations positives sur les paramètres temporels, que nous appelons aussi résistance aux anomalies d'ordonnancement, est une propriété indispensable pour la validité des algorithmes d'ordonnancement.

2.4.1 Prédictibilité de l'ordonnancement

Définition 2.6 (Algorithme prédictible)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. $\mathcal{A}$ est prédictible si $S(J_i^-(p)) \leq S(J_i(p)) \leq S(J_i^+(p))$ et $F(J_i^-(p)) \leq F(J_i(p)) \leq F(J_i^+(p))$ pour tout p tel que $1 \leq p \leq k$ et pour tout ensemble faisable de k instances $J_i^+(p)$.*

La prédictibilité est une propriété importante pour un algorithme d'ordonnancement. En effet, il semble intuitif que le pire scénario d'ordonnancement pour un ensemble d'instances se produisent lorsque chaque instance est exécutée pour une durée égale à son WCET. Pourtant, Ha et Liu ont prouvé que pour certaines approches d'ordonnancement multiprocesseur, cela n'était pas le cas [HL94].

Cucu et Goossens [CG06] ont prouvé qu'un ordonnancement multiprocesseur non oisif et conduit par les priorités est prédictible, de manière générale, pour les systèmes à processeurs uniformes. Les systèmes à processeurs identiques étant un cas particulier des systèmes à processeurs uniformes, ce résultat s'étend donc au cas des systèmes à processeurs identiques.

2.4.2 Viabilité de l'ordonnancement

Baruah et Burns ont étendu la notion de prédictibilité à la notion de viabilité (de l'anglais *sustainability*) [BB06, BB08]. Cette notion plus générale ne considère plus seulement les diminutions de durée d'exécution, mais aussi les augmentations de période,

ainsi que les augmentations d'échéance. Dans ces travaux, Baruah et Burns ne s'intéressent non pas aux algorithmes d'ordonnancement mais aux tests d'ordonnançabilité. La définition 2.7 est une autre formulation de la définition 2.6 appliquée aux tests d'ordonnançabilité.

Définition 2.7 (Test résistant aux diminutions de durée d'exécution)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement, et $\mathcal{F}$ un test d'ordonnançabilité pour $\mathcal{A}$. Soit τ un ensemble de tâches considéré ordonnançable d'après $\mathcal{F}$. Soit $\mathcal{J}$ un ensemble d'instances obtenu à partir de τ. $\mathcal{F}$ est résistant aux diminutions de durée d'exécution si et seulement si il considère ordonnançable n'importe quel ensemble d'instances obtenu à partir de $\mathcal{J}$ en diminuant la durée d'exécution d'une ou de plusieurs instances.*

Définition 2.8 (Test résistant aux augmentations d'échéance)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement, et $\mathcal{F}$ un test d'ordonnançabilité pour $\mathcal{A}$. Soit τ un ensemble de tâches considéré ordonnançable d'après $\mathcal{F}$. Soit $\mathcal{J}$ un ensemble d'instances obtenu à partir de τ. $\mathcal{F}$ est résistant aux augmentations d'échéance si et seulement si il considère ordonnançable n'importe quel ensemble d'instances obtenu à partir de $\mathcal{J}$ en augmentant l'échéance absolue d'une ou de plusieurs instances.*

Définition 2.9 (Test résistant aux augmentations de période)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement, et $\mathcal{F}$ un test d'ordonnançabilité pour $\mathcal{A}$. Soit τ un ensemble de tâches considéré ordonnançable d'après $\mathcal{F}$. Soit $\mathcal{J}$ un ensemble d'instances obtenu à partir de τ. $\mathcal{F}$ est résistant aux augmentations de période si et seulement si il considère ordonnançable n'importe quel ensemble d'instances obtenu à partir de $\mathcal{J}$ en retardant l'activation d'une ou de plusieurs instances.*

Définition 2.10 (Test viable)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement, et $\mathcal{F}$ un test d'ordonnançabilité pour $\mathcal{A}$. Soit τ un ensemble de tâches considéré ordonnançable d'après $\mathcal{F}$. Soit $\mathcal{J}$ un ensemble d'instances obtenu à partir de τ. $\mathcal{F}$ est viable si et seulement si il est résistant (i) aux diminutions de durée d'exécution, (ii) aux augmentations d'échéance et (iii) aux augmentations de période.*

Baker et Baruah se sont pour leur part intéressés aux algorithmes d'ordonnancement et à leur viabilité [BB09b].

Définition 2.11 (Algorithme résistant aux diminutions de durée d'exécution)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. Soit τ un ensemble de tâches ordonnançable avec $\mathcal{A}$. Soit $\mathcal{J}$ un ensemble d'instances obtenu à partir de τ. $\mathcal{A}$ est résistant aux diminutions de durée d'exécution si et seulement si il peut ordonnancer sans qu'aucune échéance ne soit ratée n'importe quel ensemble d'instances obtenu à partir de $\mathcal{J}$ en diminuant la durée d'exécution d'une ou de plusieurs instances.*

Définition 2.12 (Algorithme résistant aux augmentations d'échéance)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. Soit τ un ensemble de tâches ordonnançable avec $\mathcal{A}$. Soit $\mathcal{J}$ un*

ensemble d'instances obtenu à partir de τ. $\mathcal{A}$ est résistant aux augmentations d'échéance si et seulement si il peut ordonnancer sans qu'aucune échéance ne soit ratée n'importe quel ensemble d'instances obtenu à partir de $\mathcal{J}$ en augmentant l'échéance absolue d'une ou de plusieurs instances.

Définition 2.13 (Algorithme résistant aux augmentations de période)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. Soit τ un ensemble de tâches ordonnançable avec $\mathcal{A}$. Soit $\mathcal{J}$ un ensemble d'instances obtenu à partir de τ. $\mathcal{A}$ est résistant aux augmentations de période si et seulement si il peut ordonnancer sans qu'aucune échéance ne soit ratée n'importe quel ensemble d'instances obtenu à partir de $\mathcal{J}$ en retardant l'activation d'une ou de plusieurs instances.*

Définition 2.14 (Algorithme viable)**.** *Soit $\mathcal{A}$ un algorithme d'ordonnancement. Soit τ un ensemble de tâches ordonnançable avec $\mathcal{A}$. Soit $\mathcal{J}$ un ensemble d'instances obtenu à partir de τ. $\mathcal{A}$ est viable si et seulement si il est résistant (i) aux diminutions de durée d'exécution, (ii) aux augmentations d'échéance et (iii) aux augmentations de période.*

2.5 Conclusion

Nous avons présenté dans la section 2.3 deux approches pour calculer les marges sur ce que nous appelons des variations négatives sur les paramètres temporels. Ces variations sur les paramètres tendent à rendre l'ordonnancement plus difficile. Concrètement, nous avons porté notre attention sur l'augmentation des WCET et sur les réductions de périodes. La première approche (calcul d'*Allowance*) ne permettant que de calculer les marges sur le WCET, nous avons proposé son extension au domaine des périodes. Dans la section 2.4, nous avons introduit la notion de variations positives sur les paramètres temporels. Ces variations sur les paramètres, bien qu'elles tendent à rendre intuitivement l'ordonnancement plus facile, peuvent poser des problèmes d'anomalies. Nous avons donc présenté la notion de prédictibilité d'un algorithme d'ordonnancement, ainsi qu'une notion plus générale qu'est la viabilité d'un algorithme d'ordonnancement et des tests associés. La prédictibilité étant propre aux durées d'exécution des tâches, la viabilité est une extension de cette notion aux domaines des périodes et des échéances.

CHAPITRE 3

ALLOCATION DE TÂCHES INDÉPENDANTES

3.1 Introduction

Dans ce chapitre, nous étudions les algorithmes appliqués aux tâches avant le démarrage de l'ordonnancement. Il s'agit d'une approche par partitionnement (FTII) où nous considérons que l'ordonnancement en ligne est réalisé par un algorithme de la classe FTP. Nous considérons des *tâches sporadiques* à *échéan-ces contraintes*. Ces tâches sont indépendantes dans le sens où elles ne partagent pas de ressources communes.

L'approche d'ordonnancement temps réel multiprocesseur hors-ligne la plus représentative est l'approche par partitionnement. Chaque sous-ensemble de tâches obtenu à partir d'un partitionnement est ensuite ordonnancé de manière indépendante sur chaque processeur. Cette approche interdit donc les migrations inter-processeurs. Mais nous pouvons tout à fait concevoir des approches d'ordonnancement pour lesquels des schémas de migrations sont connus à l'avance.

Nous proposons un algorithme de partitionnement qui soit robuste aux variations négatives de WCET et de périodes. Plus particulièrement, un algorithme pour lequel la marge sur le WCET ou la période des tâches est maximisée. L'aspect de la robustesse qui est ici considéré consiste à rendre le système plus tolérant à des fautes, à de mauvaises estimations de WCET ou à un mauvais dimensionnement au niveau des périodes.

Dans la section 3.2, nous décrivons les modèles et les notations utilisés dans ce chapitre. Dans la section 3.3, nous présentons la problématique de l'ordonnancement par partitionnement. Dans la section 3.5, nous discutons des propriétés de robustesse des algorithmes de partitionnement. Enfin, nous proposons une synthèse de ce chapitre dans la section 3.6.

3.2 Notations

Notation	Définition
Π	Un ensemble de processeurs
π_j	Le processeur d'indice j
τ	Un ensemble de tâches
τ_i	La tâche d'indice i
$\tau(\Pi)$	L'ensemble des tâches assignées sur un ensemble de processeurs Π
$\tau(\pi_j)$	L'ensemble des tâches assignées sur le processeur π_j

TABLE 3.1 – Notations employées dans le chapitre 3.

La table 3.1 contient la synthèse des notations utilisées dans ce chapitre.

3.3 Partitionnement

Nous nous intéressons dans cette section au modèle de tâches indépendantes. L'approche par partitionnement consiste à subdiviser un ensemble de tâches en autant de sous-ensembles disjoints qu'il y a de processeurs. Chaque sous-ensemble est ensuite ordonnancé sur un processeur, et ce sans migration. Cette approche a l'avantage qu'une fois le partitionnement connu, les résultats provenant de l'état de l'art sur l'ordonnancement temps réel monoprocesseur sont valides. La difficulté dans l'étude de l'ordonnancement par partitionnement réside dans la recherche de partitions ordonnançables.

3.3.1 Anomalie de partitionnement

Certains algorithmes de partitionnement peuvent être sujets à des anomalies. Andersson et Jonsson montrent dans [AJ02] que des anomalies peuvent se produire avec l'algorithme *R-BOUND-MP* [LMM98] lorsque les périodes des tâches sont augmentées. Ce phénomène se produit car la condition d'admission *R-BOUND* (utilisée par *R-BOUND-MP*) exploite le rapport entre la période maximale et la période minimale des tâches à assigner. Une période plus grande implique un rapport différent qui peut modifier la condition d'admission. Un ensemble de tâches pouvant être partitionné par *R-BOUND-MP* peut ne plus pouvoir l'être avec une ou plusieurs tâches ayant des périodes plus grandes.

Dans le cas où le partitionnement est effectué hors-ligne, le problème des anomalies n'est qu'un problème de dimensionnement du système. En effet, le fait qu'un ensemble de tâches ne puisse pas être partitionné avec des périodes plus grandes ne conduit pas à ce que des échéances soient dépassées.

3.4 Algorithmes de partitionnement

Nous considérons une plate-forme multiprocesseur composée de processeurs identiques. Étant donné un ensemble de tâches sporadiques caractérisées par leur utilisation, nous pouvons nous poser la question de savoir combien de processeurs sont nécessaires pour pouvoir ordonnancer cet ensemble de tâches. Cette formulation du problème est très similaire à celle du problème de BIN-PACKING.

Définition 3.1 (Problème de BIN-PACKING)**.** *Étant donné des boîtes de taille V et un ensemble d'objets de taille inférieure ou égale à V à ranger dans les boîtes, le problème consiste à trouver le nombre minimum de boîtes nécessaires pour ranger les objets.*

Il est aisé de remarquer que nous pouvons transformer une instance de notre problème de partitionnement d'un ensemble de tâches en une instance du problème BIN-PACKING. Il suffit de considérer nos processeurs comme étant des boîtes dont la taille serait la puissance de calcul des processeurs et de considérer nos tâches comme des objets à ranger dont la taille serait l'utilisation de ces tâches. Nous pouvons maintenant résoudre notre problème de partitionnement en utilisant un algorithme permettant de résoudre le problème de BIN-PACKING. Malheureusement, il a été montré que ce dernier est un problème *NP-difficile* [GJ79]. Cela signifie qu'à moins qu'il soit prouvé que $P = NP$, aucun algorithme ne peut garantir de trouver la solution optimale à ce problème en temps polynomial. Une solution optimale correspondrait, pour notre problème, au nombre minimal de processeurs nécessaires pour pouvoir ordonnancer un ensemble de tâches. Heureusement, le problème de BIN-PACKING a été largement étudié et il existe de nombreuses heuristiques permettant de le résoudre sans garantie de trouver la solution optimale.

Une autre question que nous pouvons nous poser est de savoir si il existe un partitionnement d'un ensemble de tâches pour une plate-forme multiprocesseur donnée, en d'autres termes lorsque le nombre de processeurs est fixé. Ce problème correspond au problème de décision associé au problème BIN-PACKING. La description du problème donnée dans la définition 3.1 correspond au problème de minimisation. Pour résoudre le problème de décision, il suffit d'appliquer un algorithme résolvant le problème de minimisation et de vérifier que la valeur de la solution retournée est inférieure ou égale au nombre de processeurs.

Bien que nous n'ayons pas à notre disposition d'algorithme garantissant de trouver les solutions optimales en temps polynomial, la littérature concernant le problème BIN-PACKING contient plusieurs algorithmes donnant de bons résultats en temps polynomial. Naturellement, la communauté de l'ordonnancement temps réel s'est inspirée de ces derniers pour proposer des heuristiques de partitionnement.

3.4.1 Algorithmes pour BIN-PACKING dans le cas du partitionnement

Nous comparons ici les différents algorithmes utilisés pour résoudre le problème BIN-PACKING dans le but de choisir celui qui permettra de fournir des partitions les plus robustes.

```
   Entrées : τ                                  /* Un ensemble de tâches */
   Sorties : k                                /* Un nombre de processeurs */
1  Π ← π_1;
2  k ← 1;
3  pour τ_i ∈ τ faire
4      pour j ∈ {1, ..., k} faire                        /* Ordre croissant */
5          si τ_i est ordonnançable sur π_j alors
6              τ(π_j) ← τ(π_j) ∪ τ_i;
7              sortie de boucle;
8          fin
9      fin
10     si τ(Π) ∩ τ_i = ∅ alors                        /* τ_i n'est pas assignée */
11         k ← k + 1;
12         Π ← Π ∪ π_k;                              /* Ajout d'un processeur */
13         τ(π_k) ← τ(π_k) ∪ τ_i;                                /* Allocation */
14     fin
15 fin
16 retourner k
```

Algorithme 3.1: FF

First-Fit L'algorithme *First-Fit (FF)* associe un ordre à l'ensemble des tâches. Chaque tâche est ensuite assignée sur le premier processeur pour lequel le test d'ordonnançabilité réussit. Ce test dépend de l'algorithme d'ordonnancement choisi pour ordonnancer les ensembles de tâches sur les processeurs. Les processeurs sont considérés dans l'ordre croissant de leur indice. Si une tâche ne peut être assignée à aucun processeur, alors un processeur est ajouté à l'ensemble des processeurs et la tâche y est assignée.

Last-Fit L'algorithme *Last-Fit (LF)* est similaire à FF à la différence que les processeurs sont pris dans l'ordre décroissant de leur indice.

Next-Fit L'algorithme *Next-Fit (NF)* se comporte comme FF, mais avec la particularité qu'un processeur n'ayant pu admettre une tâche n'est plus considéré pour l'affectation. Cet algorithme a donc une complexité en temps réduite par rapport aux autres car pour chaque tâche, un seul processeur est considéré.

Best-Fit L'algorithme *Best-Fit (BF)* se comporte comme FF mais les processeurs sont considérés dans un ordre associé à une métrique. Dans notre cas, la métrique corres-

```
   Entrées : τ                                  /* Un ensemble de tâches */
   Sorties : k                                /* Un nombre de processeurs */
1  Π ← π_1;
2  k ← 1;
3  pour τ_i ∈ τ faire
4  |  si τ_i n'est pas ordonnançable sur π_k alors
5  |  |  k ← k + 1;
6  |  |  Π ← Π ∪ π_k;                         /* Ajout d'un processeur */
7  |  fin
8  |  τ(π_k) ← τ(π_k) ∪ τ_i;                                /* Allocation */
9  fin
10 retourner k
```

Algorithme 3.2: NF

```
   Entrées : τ                                  /* Un ensemble de tâches */
   Sorties : k                                /* Un nombre de processeurs */
1  Π ← π_1;
2  k ← 1;
3  pour τ_i ∈ τ faire
4  |  {1', ..., k'} ← decrcost({1, ..., k};                /* Coût décroissant */
5  |  pour j ∈ {1', ..., k'} faire
6  |  |  si τ_i est ordonnançable sur π_j alors
7  |  |  |  τ(π_j) ← τ(π_j) ∪ τ_i;                             /* Allocation */
8  |  |  |  sortie de boucle;
9  |  |  fin
10 |  fin
11 |  si τ(Π) ∩ τ_i = ∅ alors                          /* τ_i n'est pas assignée */
12 |  |  k ← k + 1;
13 |  |  Π ← Π ∪ π_k;                         /* Ajout d'un processeur */
14 |  |  τ(π_k) ← τ(π_k) ∪ τ_i;                             /* Allocation */
15 |  fin
16 fin
17 retourner k
```

Algorithme 3.3: BF

pond à l'utilisation de l'ensemble des tâches assignées sur le processeur. C'est-à-dire que le premier processeur testé, pour l'affectation d'une tâche, est le processeur le plus chargé. Mais ce n'est qu'un exemple et d'autres fonctions de coût peuvent être utilisées. La fonction `decrcost()` (algorithme 3.3, ligne 4) retourne l'ensemble des indices des processeurs triés par ordre de leur coût décroissant.

Worst-Fit L'algorithme *Worst-Fit (WF)* se comporte comme BF mais les processeurs sont considérés cette fois-ci dans l'ordre de leur utilisation croissante. C'est-à-dire que le premier processeur, sur lequel la tâche à assigner est testée, est le processeur le moins

chargé. Comme dans le cas de *Best-Fit*, une autre fonction de coût peut être utilisée.

```
   Entrées : τ                                  /* Un ensemble de tâches */
   Sorties : k                                /* Un nombre de processeurs */
 1 Π ← π_1;
 2 k ← 1;
 3 pour τ_i ∈ τ faire
 4 |  {1', ..., k'} ← iu({1, ..., k};                /* Util. croissante */
 5 |  pour j ∈ {2', 1', ..., k'} faire
 6 |  |  si τ_i est ordonnançable sur π_j alors
 7 |  |  |  τ(π_j) ← τ(π_j) ∪ τ_i;                          /* Allocation */
 8 |  |  |  sortie de boucle;
 9 |  |  fin
10 |  fin
11 |  si τ(Π) ∩ τ_i = ∅ alors                    /* τ_i n'est pas assignée */
12 |  |  k ← k + 1;
13 |  |  Π ← Π ∪ π_k;                            /* Ajout d'un processeur */
14 |  |  τ(π_k) ← τ(π_k) ∪ τ_i;                          /* Allocation */
15 |  fin
16 fin
17 retourner k
```

Algorithme 3.4: AWF

Almost-Worst-Fit L'algorithme *Almost-Worst-Fit (AWF)* se comporte comme WF dans le sens où les processeurs sont considérés dans l'ordre de leur utilisation croissante. La différence avec ce dernier est que le premier processeur testé pour l'affectation d'une tâche est le deuxième processeur le moins chargé.

3.4.2 Algorithmes modifiés

Les algorithmes proposés précédemment permettent de trouver une solution à un problème d'optimisation. En effet, pour un ensemble de tâches donné, ces algorithmes renvoient un nombre de processeurs nécessaires pour pouvoir les ordonnancer. Ce type de problème est adapté dans le cas du dimensionnement d'un système. Par contre, dans le cas d'une analyse d'ordonnançabilité sur une plate-forme dont le nombre de processeurs est fixé, c'est une solution au problème de décision qui doit être trouvée. Il s'agit de décider si l'ensemble de tâches peut être partitionné sur m processeurs. La solution à ce problème repose sur la solution au problème d'optimisation. Il suffit d'appliquer ce dernier et de vérifier que $k \leq m$. Dans notre étude, nous voulons maximiser la marge des tâches. Notre objectif est donc une répartition au mieux des tâches sur les différents processeurs. Malheureusement, les algorithmes présentés précédemment ne sont pas spécialement adaptés pour ce genre de comportement. En effet, il ne peuvent pas répartir les tâches sur un ensemble de m processeurs car ils procèdent pour ajout

itératif de processeur au cours du déroulement de l'algorithme. C'est pourquoi nous proposons des variantes des algorithmes WF et AWF. Ces variantes considèrent en entrée, en plus de l'ensemble des tâches, un ensemble de processeurs.

```
   Entrées : τ                                  /* Un ensemble de tâches */
   Sorties : Π                                /* Un ensemble de processeurs */
1  m ← |Π|;
2  pour τ_i ∈ τ faire
3      {π_1', ..., π_m'} ← iu(Π);                          /* Util. croissante */
4      pour π_j ∈ {π_1', ..., π_m'} faire
5          si τ_i peut être assignée sur π_j alors
6              τ(π_j) ← τ(π_j) ∪ τ_i;
7              sortie de boucle;
8          fin
9      fin
10     if τ(Π) ∩ τ_i = ∅ then                          /* τ_i n'est pas assignée */
11         retourner échec
12     end
13 fin
14 retourner succès
```

Algorithme 3.5: F-WF

Fixed-Worst-Fit L'algorithme WF choisit toujours le processeur pour lequel la fonction de coût retourne la plus petite valeur. Dans notre cas, il s'agit de l'utilisation processeur qui doit être la plus petite possible. Mais ce choix est fait parmi un ensemble de processeurs qui grandit au fur et à mesure du déroulement de l'algorithme. Finalement, on obtient une partition pour laquelle les premiers processeurs sont les plus remplis. En effet, un processeur n'est rajouté à l'ensemble que lorsque qu'une tâche ne peut être assignée sur un processeur déjà présent dans l'ensemble. Le comportement de *Fixed-Worst-Fit (F-WF)* décrit dans l'algorithme 3.5 est sensiblement le même que WF. Toutefois, les m processeurs sont considérés comme déjà présents. Cet algorithme produit une partition pour laquelle les tâches sont réparties sur tous les processeurs plutôt que concentrées sur les premiers.

Fixed-Almost-Worst-Fit L'algorithme *Fixed-Almost-Worst-Fit (F-AWF)* présente la même modification que F-WF. Pour chaque tâche, le processeur sur lequel assigner cette dernière est choisi parmi l'ensemble des m processeurs du système.

3.4.3 Ordre sur les tâches

L'ordre dans lequel les tâches sont prises pour être assignées sur les processeurs impacte grandement sur les performances de l'algorithme. Il est d'ailleurs souvent fait

```
   Entrées : τ                                  /* Un ensemble de tâches */
   Sorties : Π                                /* Un ensemble de processeurs */
1  m ← |Π|;
2  pour τ_i ∈ τ faire
3      {π_1', ..., π_m'} ← iu(Π);                          /* Util. croissante */
4      pour π_j ∈ {π_2', π_1', ..., π_m'} faire
5          si τ_i peut être assignée sur π_j alors
6              τ(π_j) ← τ(π_j) ∪ τ_i;
7              sortie de boucle;
8          fin
9      fin
10     si τ(Π) ∩ τ_i = ∅ alors                          /* τ_i n'est pas assignée */
11         retourner échec
12     fin
13 fin
14 retourner succès
```

Algorithme 3.6: F-AWF

référence à l'algorithme *First-Fit Decreasing* pour parler de l'algorithme FF où les tâches sont prises dans l'ordre décroissant de leur utilisation. Il existe différentes manières de trier les tâches par ordre de leur :

- utilisation décroissante (*Decreasing Utilization*) (DU) ;
- utilisation croissante (*Increasing Utilization*) (IU) ;
- échéances décroissantes (*Decreasing Deadline*) (DD) ;
- échéances croissantes (*Increasing Deadline*) (ID) ;
- périodes décroissantes ;
- périodes croissantes (*Increasing Period*) (IP) ;
- WCET décroissant (*Decreasing WCET*) (DW) ;
- WCET croissants (*Increasing WCET*) (IW).

Nous considérons l'ordre DU car c'est l'un des plus utilisés dans l'état de l'art.

3.4.4 Travaux existants

First-Fit L'algorithme *Rate-Monotonic-First-Fit* (RMFF) a été proposé par Dhall et Liu [DL78], puis étudié à nouveau par Oh et Son [OS93]. Il s'agit de l'un des premiers algorithmes de partitionnement basé sur un algorithme pour le problème BIN-PACKING. Il a été proposé pour partitionner des tâches périodiques à échéances implicites.

L'algorithme *First-Fit Decreasing-Utilization Factor* (FFDUF) a été proposé par Davari et Dhall [DD86].

L'algorithme *Fisher-Baruah-Backer First-Fit-Decreasing* (FBB-FFD) a été proposé par Fisher, Baruah et Baker [FBB06b]. Il est capable de partitionner un ensemble de tâches sporadiques à échéances arbitraires.

Nom	Algorithme	Ordre	Priorité	Modèle de tâche
RMFF [DL78]	FF	IP	RM	périodique / échéances implicites
RMNF [DL78]	NF	IP	RM	périodique / échéances implicites
FFDUF [DD86]	FF	DU	RM	périodique / échéances implicites
RMBF [OS93]	BF	IP	RM	périodique / échéance implicites
RM-DU-NFS [AJ02]	NF	DU	RM	périodique / échéances implicites
FBB-FFD [FBB06b]	FF	ID	DM	sporadique / échéances arbitraires

TABLE 3.2 – Algorithmes de partitionnement

Best-Fit L'algorithme *Rate-Monotonic-Best-Fit* (RMBF) [OS93] a été proposé par Oh et Son. Il est basé sur l'algorithme BF et permet de partitionner un ensemble de tâches périodiques à échéances implicites.

Next-Fit L'algorithme *Rate-Monotonic-Next-Fit* (RMNF) a été proposé par Dhall et Liu [DL78], puis étudié à nouveau par Oh et Son [OS93]. Il s'agit de l'un des premiers algorithmes de partitionnement basé sur un algorithme pour le problème BIN-PACKING. Il a été proposé pour ordonnancer des tâches périodiques à échéances implicites.

L'algorithme *Rate-Monotonic Decreasing-Utilization Next-Fit-Scheduling* (RM-DU-NFS) a été proposé par Andersson et Jonsson [AJ02]. Il a été conçu pour pallier aux anomalies de partitionnement. On parle d'anomalies lorsqu'un ensemble de tâches peut être partitionné mais qu'un ensemble de tâches avec une plus petite utilisation ne le peut pas.

Tous ces algorithmes référencées dans la table 3.2 ont été conçus dans le but d'offrir une meilleure ordonnançabilité ou d'éviter les anomalies de partitionnement dans le cas de RM-DU-NFS. Mais aucun de ces algorithmes ne prend en considération la tolérance aux dépassements de WCET ou à la réduction de période.

3.5 Partitionnement robuste

Dans cette section, nous proposons un algorithme de partitionnement dont l'objectif est de maximiser la marge des tâches. Cet algorithme est décliné en deux version :

- *Allowance-Fit-WCET* (AF^C) pour la marge sur le WCET ;
- *Allowance-Fit-Frequency* (AF^f) pour la marge sur la période.

3.5.1 Description de l'algorithme

```
   Entrées : τ                                  /* Un ensemble de tâches */
   Sorties : Π                                /* Un ensemble de processeurs */
1  m ← |Π|;
2  minMarge[m]i;
3  pour τ_i ∈ τ faire
4      pour j ∈ {1, ..., m} faire
5          minMarge[j] ← marge_minimale(τ(π_j) ∪ τ_i);
6          si max_{j=1}^{m} minMarge[j] = −1 alors
7              retourner échec
8          fin
9      fin
10     π_k ← max_{j=1}^{m} minMarge[i];
11     τ(π_k) ← τ(π_k) ∪ τ_i;
12 fin
13 retourner succès
```

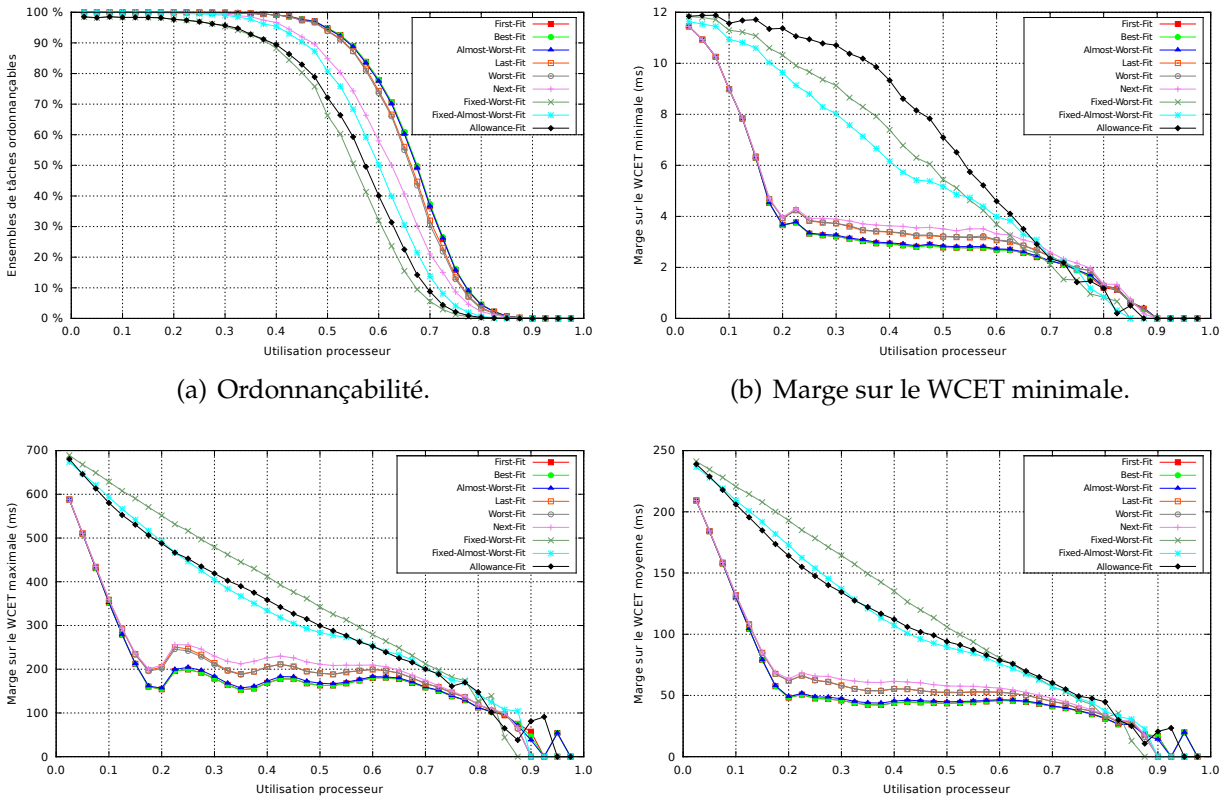
Algorithme 3.7: AF

L'algorithme 3.7 décrit le comportement général d'*Allowance-Fit (AF)*. La distinction entre AF^C et AF^f est faite par la fonction de calcul de marge (marge sur le WCET ou marge sur la période).

Le principe de cet algorithme est d'allouer les tâches sur le processeur où la marge minimale est la plus grande. Pour chaque processeur π_j (ligne 4-9), cette marge minimale est calculée (ligne 5) en considérant que la tâche τ_i est assignée sur π_j. La fonction `marge_minimale()` appelée avec la valeur j renvoit la valeur -1 si au moins une tâche n'est pas ordonnançable sur π_j. Si τ_i ne peut être assignée sur aucun processeur, l'algorithme échoue (ligne 6-8). Dans le cas contraire, le processeur choisit pour l'allocation de τ_i est celui ayant la plus grande valeur de marge minimale (ligne 10-11). Si toutes les tâches ont pu être allouées, l'algorithme termine avec succès (ligne 13).

3.5.2 Protocole de simulation

Les résultats de simulation présentés dans cette section ont été produits par notre simulateur. Chaque simulation se compose de 10000 ensembles de tâches pour chaque valeur d'utilisation parmi $[0.025, 0.05, \ldots, 0.975] \times m$ où m est le nombre de processeurs. Une simulation revient donc à l'analyse de 390000 ensembles de tâches engendrés aléatoirement. Les ensembles de tâches sont constitués de 16 tâches à échéances contraintes et la plate-forme multiprocesseur est composée de 4 processeurs.



(a) Ordonnançabilité.

(b) Marge sur le WCET minimale.

(c) Marge sur le WCET maximale.

(d) Marge sur le WCET moyenne.

FIGURE 3.1 – Algorithmes de partitionnement avec utilisation décroissante, ordonnancement Deadline-Monotonic et analyse de temps de réponse.

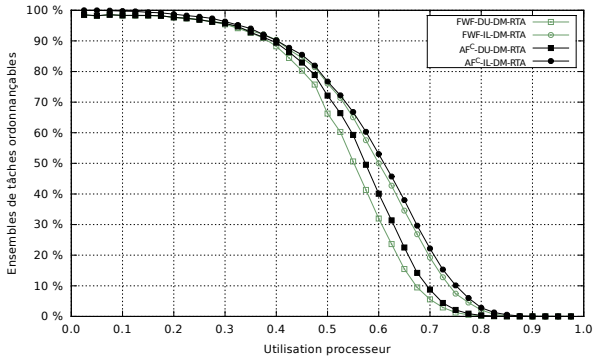
3.5.3 Marge sur le WCET

Dans cette section, nous appelons « marge » la marge sur le WCET. La figure 3.1 représente la comparaison d'un ensemble d'algorithmes de partitionnement. L'ordre sur les tâches, l'algorithme d'attribution des priorités, ainsi que le test d'ordonnançabilité sont les mêmes. Ainsi, pour tous les ensembles de tâches, les tâches sont triées dans l'ordre DU, les priorités sont attribuées suivant l'algorithme DM et l'acceptation d'une tâche sur un processeur relève d'une analyse de temps de réponse. Ainsi, ce sont les performances des algorithmes pour BIN-PACKING qui sont comparées et non pas les performances de conditions suffisantes d'ordonnançabilité. Dans la figure 3.1(a), nous comparons les performances en terme d'ordonnançabilité des algorithmes de partitionnement. Nos résultats sont concordants avec ceux que nous pouvons trouver dans la littérature, à savoir que les algorithmes FF et BF sont les plus performants en terme d'ordonnançabilité. Nous remarquons que l'algorithme AWF, qui, à la différence de FF et BF est très rarement cité dans la littérature, offre des performances en terme d'ordonnançabilité équivalentes. Les algorithmes LF et WF sont les algorithmes qui appliquent le comportement inverse des algorithmes FF et BF. Alors qu'avec FF c'est le premier processeur qui peut accepter une tâche qui est choisi, avec LF, c'est le dernier. Bien que LF et WF peuvent sembler être les algorithmes antagonistes de FF et BF, ils offrent des per-

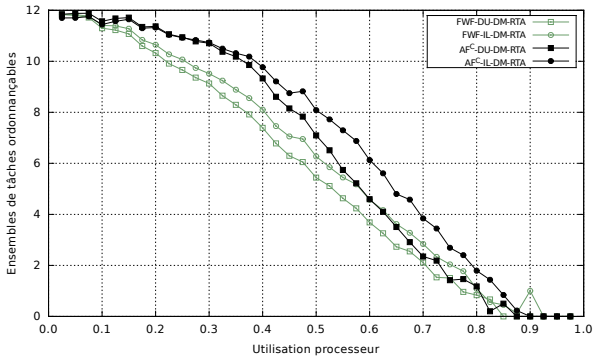
formances qui sont assez proches. L'algorithme NF est moins performant car il bloque l'accès à un processeur dès lors qu'une tâche ne peut y être admise. Cette propriété réduit l'ordonnançabilité mais présente l'avantage d'empêcher les anomalies de partitionnement. Alors que les choix faits par BF, WF et AWF sont souvent les mêmes, ceux de F-WF et de F-AWF sont assez différents. En effet, BF, WF et AWF commencent le partitionnement en ne considérant qu'un seul processeur pour l'allocation des tâches. Si une allocation échoue, alors un processeur est ajouté tant que le nombre de processeurs ne dépasse pas celui de la plate-forme cible. En revanche, F-WF et F-AWF calculent un partitionnement directement sur un ensemble de processeurs correspondant au nombre de processeurs de la plate-forme considérée. Enfin, nous avons proposé un algorithme de partitionnement, nommé AF^C spécialement conçu pour maximiser la marge minimale des tâches. Bien qu'il soit plus complexe en terme de temps de calcul que F-WF, il offre de meilleures performances en terme d'ordonnançabilité.

Dans la figure 3.1(b), nous représentons la marge minimale (définie dans la section 2.3.1) obtenue à partir de chacun des algorithmes présentés précédemment. La marge minimale est la plus petite valeur de marge qui peut être ajoutée à une tâche. Elle nous donne un indicateur sur la robustesse aux dépassements de WCET du système. Nous montrons dans cette figure que l'algorithme AF^C offre la meilleure alternative pour maximiser la valeur de marge minimale pour des valeurs d'utilisation comprises dans l'intervalle $[0.025, 0.065] \times m$ où m est le nombre de processeurs. Pour des valeurs d'utilisation supérieures, les systèmes sont trop chargés pour pouvoir établir la prédominance d'un algorithme par rapport aux autres. La complexité d'AF^C est polynomiale en le nombre de tâches. Les algorithmes F-WF et F-AWF restent toutefois des alternatives intéressantes car moins complexes à implanter. Leur complexité, indépendamment du test d'ordonnançabilité, est linéaire en le nombre de tâches. Par contre, tous les algorithmes couramment utilisés dans la littératures, qui sont à l'origine des algorithmes fondés sur BIN-PACKING, ont des résultats bien en deçà des 3 algorithmes précédemment cités. Nous représentons dans la figure 3.1(c), et respectivement dans la figure 3.1(b), la marge maximale, et respectivement la marge moyenne, obtenue pour les ensembles de tâches testés. Les courbes représentées nous permettent de confirmer que les 3 meilleurs algorithmes de la figure 3.1(b) surpassent bien les autres en général, et pas seulement dans le cas de la marge minimale. Bien que AF^C soit prépondérant dans le cas de la marge minimale, ce n'est pas le cas pour la marge maximale et moyenne. Ce qui signifie que l'algorithme F-WF reste une bonne alternative pour fournir un partitionnement robuste aux dépassements de WCET.

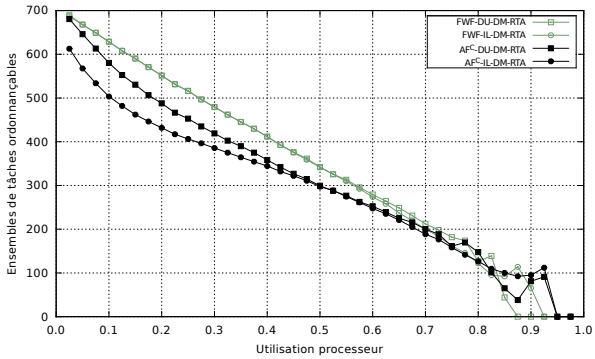
Dans la figure 3.1, nous avons comparé les algorithmes de placement des tâches sur les processeurs (algorithmes fondés sur BIN-PACKING). Nous nous intéressons maintenant à l'influence de l'ordre sur les tâches. Jusqu'à maintenant, nous considérions que les tâches étaient triées selon l'ordre DU. Cet ordre est parmi l'un des plus couramment utilisé, sûrement par analogie avec la taille des objets dans le problème BIN-PACKING.



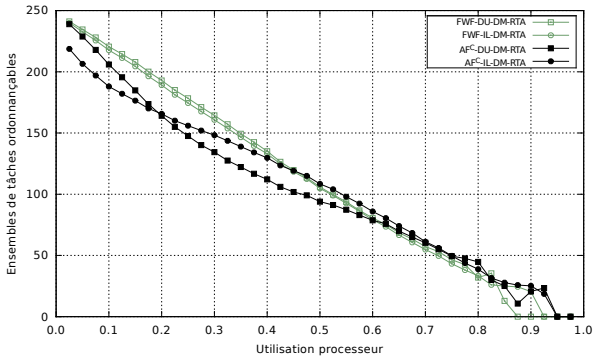
(a) Ordonnançabilité.



(b) Marge minimale sur le WCET.



(c) Marge maximale sur le WCET.



(d) Marge moyenne sur le WCET.

FIGURE 3.2 – Algorithmes de partitionnement avec Deadline-Monotonic et analyse de temps de réponse. Comparaison entre utilisation décroissante et laxité croissante.

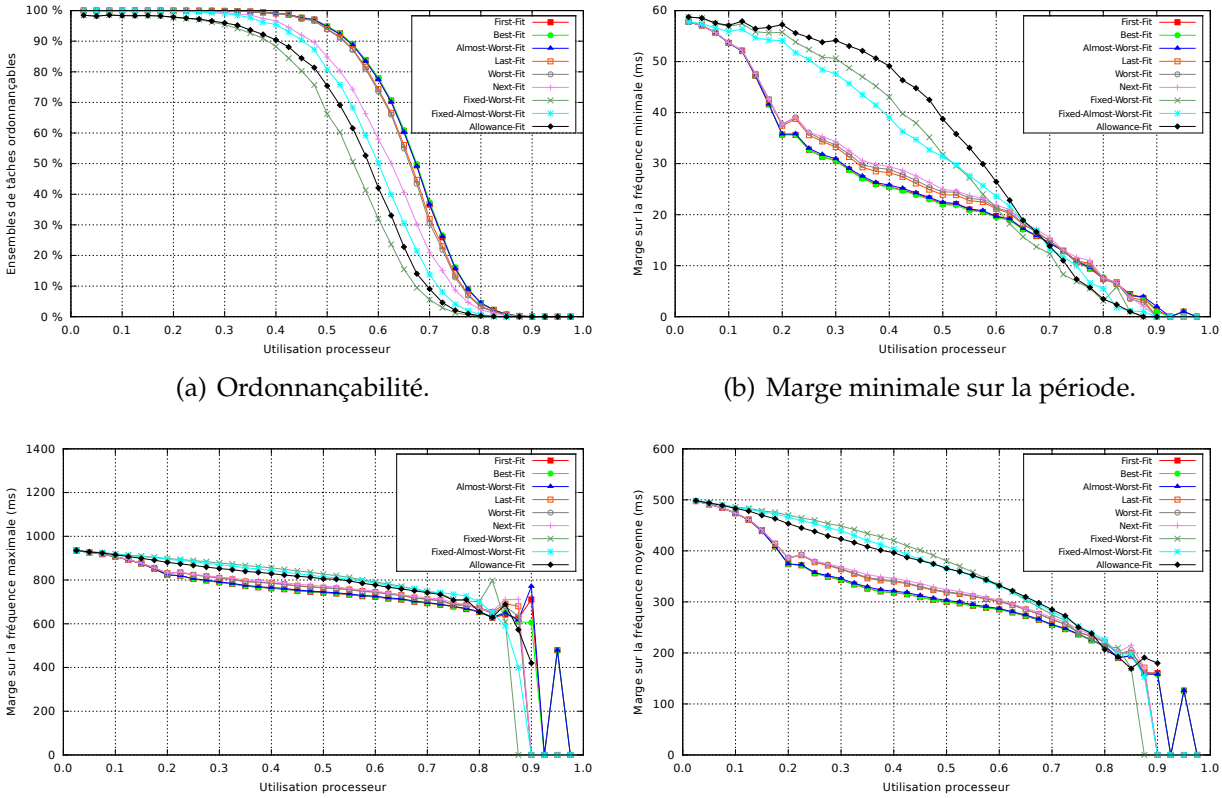
Dans le but de maximiser la marge minimale, nous proposons de trier les tâches par ordre de laxité croissante (*Increasing Laxity*) (IL). Nous définissons la laxité d'une tâche comme la durée entre la terminaison de son exécution et son échéance. Ainsi, une tâche τ_i considérée seule a comme valeur de laxité $L_i = D_i - C_i$. Cet ordre est intuitif car il conduit à placer en premier les tâches qui ont le moins de laxité.

Dans la figure 3.2, nous comparons l'ordre DU avec l'ordre IL pour les algorithmes F-WF et AF^C. Dans la figure 3.2(a), nous représentons l'ordonnançabilité des ensembles de tâches obtenue à partir ces algorithmes. Nous voyons donc que le fait de prendre les tâches dans l'ordre IL améliore légèrement l'ordonnançabilité sans ajouter aucune complexité à l'algorithme de partitionnement.

Dans la figure 3.2(b), nous voyons que l'ordre IL, améliore les performances de l'algorithme AF^C en terme de maximisation de la marge minimale. Cette amélioration est encore plus flagrante pour l'algorithme F-WF. Par contre, dans les figures 3.2(c) et 3.2(d), la marge maximale et moyenne qui peut être ajoutée aux tâches est inférieure avec l'ordre IL. Ce résultat est dû au fait que maximiser la marge minimale tend à réduire la marge maximale et moyenne. Notre objectif étant de garantir la plus grande valeur de marge pour toutes les tâches de l'ensemble de tâches, l'ordre IL apparaît être un choix intéressant. Couplé à l'algorithme AF^C, il permet ainsi d'obtenir une bonne

robustesse aux dépassements de WCET.

3.5.4 Marge sur la période



(a) Ordonnançabilité.

(b) Marge minimale sur la période.

(c) Marge maximale sur la période.

(d) Marge moyenne sur la période.

FIGURE 3.3 – Algorithmes de partitionnement avec utilisation décroissante, ordonnancement Deadline-Monotonic et analyse de temps de réponse.

Dans cette section, nous appelons « marge » la marge sur la période. Nous proposons également l'algorithme AF^f dans le but de maximiser la marge sur la période. La différence entre AF^C et AF^f est l'algorithme de calcul de marge utilisé, à savoir le calcul de marge sur le WCET pour AF^C et le calcul de marge sur la période pour AF^f. Nous l'avons comparé à l'ensemble des algorithmes auxquels nous avions comparé AF^C pour la marge sur le WCET. Dans la figure 3.3(a), nous montrons que AF^f a les mêmes performances en terme d'ordonnançabilité que AF^C. Concernant notre métrique principale, à savoir la marge minimale, nous montrons dans la figure 3.3(b) que AF^f pour les périodes est le meilleur algorithme pour une utilisation allant jusqu'à 65%. Pour une utilisation supérieure, la laxité des tâches n'est plus suffisante pour faire ressortir un algorithme prédominant. Nous remarquons qu'il est possible de faire le parallèle entre la marge minimale sur le WCET et la marge minimale. En effet, les résultats pour ces deux métriques sont sensiblement équivalents. Dans les figures 3.3(c) et 3.3(d), nous montrons les résultats concernant les valeurs maximales et moyennes de marge. Au-

cun des algorithmes ne se démarque vraiment, même si les algorithmes F-WF, F-AWF et AF^f restent les meilleurs algorithmes. Encore une fois, notre métrique la plus significative étant la marge minimale, l'algorithme AF^f reste le choix le plus intéressant pour maximiser la marge minimale.

3.6 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la classe d'ordonnancement (FTP-FTII) dans le cas d'un ordonnancement hors-ligne. Cette classe d'ordonnancement est plus connue dans la littérature sous la dénomination d'ordonnancement par partitionnement. Nous avons considéré un modèle de tâches sporadiques à échéances contraintes ne partageant pas de ressource. Nous avons montré par une étude basée sur la simulation que l'algorithme AF^C-IL-DM est un choix intéressant pour maximiser la marge minimale sur le WCET. Nous avons aussi établi le parallèle avec la marge minimale sur la période et nous avons montré que les algorithmes AF^C et AF^f sont les candidats les plus efficaces pour maximiser la marge minimale. Ce travail a donné lieu aux publications [FMG09, FMG10a].

CHAPITRE 4

ALLOCATION DE TÂCHES DÉPENDANTES

4.1 Introduction

Nous considérons dans ce chapitre une approche par partitionnement (FTII). Les tâches à ordonnancer sont des *tâches sporadiques* à *échéan-ces contraintes*.

L'état de l'art sur l'ordonnancement de tâches temps réel met en évidence un nombre important de travaux où des ensembles de tâches indépendantes sont considérés. L'hypothèse faite par ces modèles est que les tâches ne sont en concurrence que pour l'accès aux processeurs. Elles ne partagent aucune autre ressource (périphériques, mémoire, ...). L'étude de tels modèles de tâches peut paraître surprenante car elle ne reflète pas les modèles d'architectures existantes. Pourtant, les modèles de tâches indépendantes sont essentiels pour l'étude des systèmes temps réel car ils permettent de formaliser plus simplement les problèmes d'ordonnancement ou d'allocation des tâches. Le test d'ordonnançabilité pour EDF dans le cas de tâches à échéances implicites proposé par Liu et Layland ($\sum_{i=1}^{n} C_i/T_i \leq 1$) [LL73] a été exploité abondamment et l'est toujours à l'heure actuelle. Il ne l'aurait peut-être pas été si il avait dû prendre en considération les interférences dûes aux attentes sur des ressources et donc utiliser un formalisme plus complexe.

Néanmoins, une fois que les bases algorithmiques ont été posées, il convient d'étendre les modèles pour que la représentation des systèmes se rapproche au mieux des architectures existantes. Les plates-formes multiprocesseurs récentes embarquent pour la plupart des mémoires partagées, ainsi que de nombreux périphériques. Ainsi, les tâches vont pouvoir échanger des données avec le monde extérieur, mais surtout entre elles. C'est la motivation qui nous pousse à nous intéresser à des modèles de tâches partageant des données. Si une tâche est préemptée alors qu'elle utilise des données partagées, ces données peuvent se retrouver dans un état incohérent. Pour résoudre ce problème, les instructions qui utilisent des ressources partagées sont exécutées dans des sections critiques. L'entrée dans ces sections critiques est protégée par un méca-

nisme de synchronisation. Un mécanisme simple de synchronisation consiste à poser un verrou lors de l'entrée dans la section critique. Ce verrou doit être commun à toutes les sections critiques utilisant la même ressource. La prise du verrou ne peut se faire que si ce dernier est libre. Dans le cas contraire, l'instance est bloquée en attente de libération de ce verrou.

Le principe de la synchronisation est simple, il doit toutefois être manipulé avec attention. En effet, une tâche de haute priorité peut se retrouver en attente de libération d'un verrou pour une durée qui peut ne pas être bornée. Ce dernier problème, bien qu'il puisse être acceptable dans le cas d'un ordonnancement classique, pose un souci majeur dans le cas des systèmes temps réel lors de leur analyse. C'est pourquoi des protocoles de synchronisation ont été mis en place afin de définir des règles pour la prise des verrous, et ainsi éviter les inversions de priorités non bornées. En plus du protocole de synchronisation, il est nécessaire de fournir une borne sur le pire temps de blocage que peut subir une tâche. Il devient alors possible de borner le pire temps de réponse de cette tâche, et ainsi d'établir une condition d'ordonnançabilité pour un algorithme d'ordonnancement donné. Notre contribution est de proposer un algorithme de partitionnement de tâches dépendantes qui optimise la robustesse temporelle du système en maximisant la marge des tâches.

Ce chapitre est organisé de la manière suivante. Dans la section 4.2, nous définissons la terminologie et les notations utilisées dans ce chapitre. Dans la section 4.3, nous faisons l'état de l'art des principaux protocoles de synchronisation temps réel multiprocesseurs. Dans la section 4.4, nous présentons quelques approches proposées pour le problème du partitionnement de tâches dépendantes. Dans la section 4.5, nous proposons une solution de partitionnement qui optimise le critère de robustesse temporelle. Nous évaluons cette solution dans la section 4.6. La section 4.7 fait la synthèse de ce chapitre.

4.2 Terminologie et notations

4.2.1 Modèle

Dans cette étude, nous considérons que chaque tâche τ_i peut utiliser un ensemble de ressources $\mathcal{R}(\tau_i)$ qui lui sont accessibles. Lorsqu'une tâche accède à la ressource $\mathcal{R}_k$, elle entre dans une *section critique* s_k. Le WCET d'une tâche est donc composé d'un ensemble de sections d'exécution indépendante et de *sections critiques*. Les protocoles de synchronisation décrivent l'entrée dans une section critique par l'utilisation d'un *sémaphore* ou la prise d'un *verrou*. Dans le but de ne pas surcharger la description des protocoles de synchronisation, nous dirons qu'une instance *peut utiliser* une ressource lorsqu'elle est en mesure de mettre en place les mécanismes nécessaires à la synchronisation de cette ressource.

4.2.2 Notations

Notation	Définition
τ_i	La tâche d'indice i
J_i	Une instance de τ_i
Π	Un ensemble de processeurs
π_j	Le processeur d'indice j
$\mathcal{R}_k$	La ressource locale d'indice k
$\mathcal{R}_{Gk}$	La ressource globale d'indice k
$\mathcal{R}^l$	Une ressource longue
$\mathcal{R}^s$	Une ressource courte
s_k	Une section critique de $\mathcal{R}_k$
$\max(\|C(s_k)\|)$	Longueur de la plus grande section critique de $\mathcal{R}_k$
$\max(\|C(s_k, \tau_i)\|)$	Longueur de la plus grande section critique de $\mathcal{R}_k$ utilisée par τ_i
$p(J_i)$	La priorité de J_i
$\overline{p}(\mathcal{R}_k)$	La priorité plafond de $\mathcal{R}_k$
$\overline{p}(t)$	La priorité plafond du système à l'instant t
$\overline{p}(t, \pi_j)$	La priorité plafond de π_j à l'instant t
$\rho(J_1)$	Le niveau de préemption de J_1
$\rho(\mathcal{R}_k)$	Le niveau de préemption de $\mathcal{R}_k$
$\overline{\rho}(t)$	Le niveau de préemption plafond du système à l'instant t
$\overline{\rho}(t, \pi_j)$	Le niveau de préemption plafond du système à l'instant t sur π_j
B_i	Le pire temps de blocage de τ_i

TABLE 4.1 – Notations employées dans le chapitre 4.

4.3 Protocoles de synchronisation

Afin de choisir le protocole de synchronisation le plus adapté, nous avons étudié le comportement de 3 protocoles proposés dans le cadre des systèmes temps réel multiprocesseurs. Il convient de noter que l'allocation des tâches et des ressources est statique.

4.3.1 MPCP

Le protocole de synchronisation *Multiprocessor Priority Ceiling Protocol (MPCP)* a été proposé par Rajkumar [RR90]. Il est l'extension au cas des systèmes multiprocesseurs du protocole *Priority Ceiling Protocol (PCP)* proposé par Sha, Rajkumar et Lehoczky [SRL90].

Avec PCP, une priorité plafond est associée à chaque ressource partagée $\mathcal{R}_k$. Cette priorité $\overline{p}(\mathcal{R}_k)$ est définie comme la priorité de la tâche la plus prioritaire qui peut utiliser cette ressource. À l'instant t, la priorité plafond du système $\overline{p}(t)$ est donnée par la plus haute priorité plafond des ressources en cours d'utilisation (verrouillées).

À l'instant t, l'instance J_i peut utiliser une ressource $\mathcal{R}_k$ seulement si la priorité de J_i est strictement supérieure à la priorité plafond du système : $p(J_i) > \overline{p}(t)$. Dans le cas contraire, l'instance J_j qui bloque J_i hérite de sa priorité $p(J_i)$ jusqu'à ce qu'elle libère toutes les ressources de priorité plafond supérieure à $p(J_i)$.

Avec MPCP, deux types de ressources partagées sont distingués. Les ressources locales sont partagées par des tâches assignées sur le même processeur tandis que les ressources globales sont partagées par des tâches assignées sur des processeurs différents. La différence avec PCP est qu'une ressource globale $\mathcal{R}_{Gk}$ a une priorité plafond $\overline{p}(\mathcal{R}_{Gk})$ égale à la somme de la plus haute priorité du système $\overline{p}_G$ et de la priorité de la tâche la plus prioritaire qui peut utiliser cette ressource. Une instance utilisant une ressource globale est exécutée à la priorité plafond de celle-ci.

Un exemple mettant en œuvre l'utilisation de MPCP est donné dans l'annexe A.1.

4.3.2 MSRP

Le protocole de synchronisation *Multiprocessor Stack Resource Policy (MSRP)* a été proposé par Gai, Lipari et Di Natale [GLDN01]. Il est l'extension au cas des systèmes multiprocesseurs du protocole *Stack Resource Policy (SRP)* proposé par Baker [Bak91].

Chaque instance J_i possède un niveau de préemption $\rho(J_i)$. J_i ne peut préempter J_j que si $\rho(J_i) > \rho(J_j)$. Chaque ressource $\mathcal{R}_k$ possède un plafond de préemption $\rho(\mathcal{R}_k)$ qui est le maximum des niveaux de préemption des instances pouvant utiliser $\mathcal{R}_k$. À l'instant t, le plafond de préemption du système $\overline{\rho}(t)$ est donné par le plus haut plafond de préemption des ressources en cours d'utilisation.

Avec SRP, les instances actives sont placées dans une pile par ordre décroissant de leur priorité. Pour qu'une instance J_i puisse préempter une instance J_j, il faut que sa priorité soit la plus haute parmi celles des instances actives et que son niveau de préemption soit supérieur au plafond de préemption du système.

Avec MSRP, chaque processeur π_j dispose de son plafond de préemption $\overline{\rho}(t, \pi_j)$. MSRP se comporte comme SRP dans le cas des ressources locales. Pour chaque ressource globale $\mathcal{R}_{Gk}$, tous les processeurs π_j définissent un plafond supérieur ou égal au niveau de préemption maximum sur π_j. Lorsqu'une instance J_i veut utiliser une ressource globale $\mathcal{R}_{Gk}$ sur le processeur π_j, $\overline{\rho}(t, \pi_j)$ est élevé à $\rho(\mathcal{R}_{Gk})$, rendant ainsi J_i non préemptible.

Un exemple mettant en œuvre l'utilisation de MSRP est donné dans l'annexe A.2.

4.3.3 FMLP

Flexible Multiprocessor Locking Protocol (FMLP) est un protocole de synchronisation spécifiquement conçu pour les plates-formes multiprocesseurs. Il a été proposé par Block *et al.* [BLBA07]. Ce protocole peut être utilisé dans le cas d'un ordonnancement EDF par partitionnement, d'un ordonnancement EDF global, ainsi que dans le

cas de PD^2 [AS00]. Il a été étendu au cas d'un ordonnancement par partitionnement FTP par Brandenburg et Anderson [BA08]. Une des caractéristiques de ce protocole est de considérer deux types de ressources : des ressources courtes $\mathcal{R}^s$ et des ressources longues $\mathcal{R}^l$. Cette caractéristique permet de tirer parti à la fois de l'approche d'attente active (pour les ressources courtes) et à la fois de l'approche par suspension (pour les ressources longues). Une tâche bloquée en attente d'une ressource courte continuera donc son exécution de manière non préemptive empêchant ainsi les autres tâches du système de s'exécuter. En revanche, les tâches bloquées en attente d'une ressource longue seront suspendues, laissant ainsi le processeur libre pour ordonnancer d'autres tâches. Le choix de la classification des ressources (courtes ou longues) est laissé aux soins du développeur.

Lorsqu'une instance J_1 tente d'utiliser une ressource courte $\mathcal{R}_k^s$, elle devient non préemptible et est mise en attente active dans une structure *First In First Out (FIFO)*. Une fois $\mathcal{R}_k^s$ libérée et J_1 en tête de la FIFO, elle peut prendre le verrou associé à $\mathcal{R}_k^s$, utiliser la ressource de manière non préemptive, puis la libérer pour continuer son exécution de manière préemptive. Dans le cas d'une ressource longue $\mathcal{R}_k^l$, J_1 est suspendue et mise en attente passive dans une FIFO. Une fois $\mathcal{R}_k^s$ libérée et J_1 en tête de la FIFO, elle peut prendre le verrou associé à $\mathcal{R}_k^s$, utiliser la ressource de manière non préemptive, puis la libérer.

Un exemple mettant en œuvre l'utilisation de FMLP est donné dans l'annexe A.3.1.

4.4 Partitionnement de tâches dépendantes

Tindell, Burns et Wellings ont proposé un algorithme de partitionnement de tâches dépendantes [TBW92]. Ils ont considéré le cas d'une architecture distribuée où la synchronisation est garantie par un protocole d'échange de messages. Leur approche est basée sur l'algorithme de *recuit simulé*, que nous détaillons dans la section 4.5.1 qui lui est dédiée.

Le *recuit simulé* a également été utilisé par Di Natale et Stankovic pour minimiser les décalages d'activations dans les systèmes temps réel multiprocesseurs distribués [DNS95]. Cette approche offrant de bons résultats en terme de maximisation de l'ordonnançabilité, nous l'avons étendue dans le but de maximiser la marge des tâches.

Lakshmanan, de Niz et Rajkumar ont récemment proposé l'algorithme de partitionnement *Synchronization-Aware Partitioning Algorithm (SPA)* pour les tâches dépendantes [LdNR09]. Cette heuristique s'appuie sur le protocole de synchronisation MPCP.

Le principe de cet algorithme est de rassembler sur le même processeur les tâches partageant des ressources. Dans un premier temps, les tâches partageant des ressources sont regroupées. Ce regroupement est transitif, ce qui signifie que si une tâche τ_1 partage une ressource avec une tâche τ_2, et que τ_2 partage une ressource avec une tâche τ_3, ces 3 tâches sont dans le même groupe $\mathcal{G}$. Les tâches d'un groupe sont triées dans

l'ordre DU. L'algorithme commence le partitionnement avec autant de processeurs qu'il est nécessaire pour assigner l'ensemble des tâches (c'est-à-dire que l'algorithme commence avec 4 processeurs pour un ensemble de tâches d'utilisation 3.5).

Les groupes de tâches et les tâches indépendantes sont ordonnés suivant leur utilisation décroissante (ordre DU). L'algorithme tente d'assigner les groupes de tâches sur les processeurs avec BF en utilisant le critère d'utilisation processeur des groupes de tâches. Aucun processeur n'est ajouté à ce stade de l'algorithme et les groupes ne pouvant être assignés entièrement sur un processeur sont stockés dans une liste d'attente.

Les groupes restants sont triés par ordre croissant d'un coût de pénalité. Ce coût de pénalité correspond au temps de blocage induit par la transformation des ressources locales en ressources globales. Le coût de pénalité d'un groupe $cost(\mathcal{G})$ est défini de la manière suivante :

$$cost(\mathcal{G}) = \sum_{\mathcal{R}_k \in \mathcal{R}(\mathcal{G})} cost(\mathcal{R}_k)$$

où $\mathcal{R}(\mathcal{G})$ est l'ensemble des ressources partagées par les tâches de $\mathcal{G}$. Le coût de pénalité d'une ressource $cost(\mathcal{R}_k)$ est donné par :

$$cost(\mathcal{R}_k) = cost^G(\mathcal{R}_k) - cost^L(\mathcal{R}_k)$$

avec

$$cost^G(\mathcal{R}_k) = \max(|C(s_k)|)/\min_i(p(\tau_i)$$

et

$$cost^L(\mathcal{R}_l) = \max_{\tau_i \in \tau(\mathcal{R}_k)} (\max(|C(s_k, \tau_i)|/p(\tau_i)))$$

Les tâches du groupe ayant le plus petit coût de pénalité sont assignées sur les processeurs. Les tâches sont prises dans l'ordre DU et allouées avec WF (sur le processeur le moins chargé). Si une tâche ne peut pas être placée, un processeur est ajouté et l'algorithme se poursuit jusqu'à ce que toutes les tâches aient été assignées.

4.5 Partitionnement robuste

Dans cette section, nous présentons l'algorithme de partitionnement *Robust Partitioning based on Simulated Annealing (RPSA)* basé sur le *recuit simulé* [FMG10b]. Il a été proposé dans le but de maximiser la marge des tâches pour accroître la robustesse temporelle.

Après avoir étudié les différents protocoles de synchronisation temps réel multiprocesseur, nous avons fait le choix d'utiliser FMLP. Ce choix est justifié par le fait que FMLP exploite deux mécanismes de blocages :

- l'attente active pour les ressources courtes. Ce mécanisme, consistant à rendre l'instance bloquée non préemptible, est exploité par MPCP ;

– l'attente passive pour les ressources longues. Ce mécanisme, consistant à interrompre l'exécution de l'instance, est exploité par MSRP.

FMLP est donc un protocole flexible tirant parti des avantages des protocoles MPCP et MSRP. Nous différencions les ressources courtes des ressources longues de la manière suivante. Si la longueur maximale des sections critiques d'une ressource $\mathcal{R}_k$ est inférieure ou égale à une borne fixée [1], alors $\mathcal{R}_k$ est une ressource courte. $\mathcal{R}_k$ est une ressource longue dans le cas contraire.

4.5.1 Recuit simulé

L'algorithme de *recuit simulé* est une méta-heuristique proposée par Kirkpatrick, Gelatt et Vecchi [KGV83]. Une méta-heuristique est un algorithme permettant de résoudre des problèmes d'optimisation difficiles lorsque les algorithmes permettant de résoudre ces problèmes ne sont pas connus ou sont trop complexes. Le *recuit simulé* s'inspire d'un processus utilisé en métallurgie pour obtenir l'état le plus stable d'un métal et donc accroître sa solidité. Ce processus consiste à contrôler le refroidissement du métal en alternant des cycles de refroidissement lents et des cycles de réchauffage (recuits). Le comportement de l'algorithme de *recuit simulé* est fondé sur ce principe. Une instance du problème d'optimisation, engendrée aléatoirement, sert de donnée d'entrée. Cette instance du problème, par analogie avec le recuit métallurgique, correspond à un état instable du système. À partir de cet état instable, un autre état du système est obtenu en appliquant des modifications aléatoires. La notion d'énergie du système intervient alors, et permet de faire converger le système vers un état stable, c'est-à-dire vers une solution qui se rapproche de l'optimale. Si l'énergie du système engendré est inférieure à l'énergie du système d'origine, le système engendré est conservé. Sinon, un tirage aléatoire permet de décider si il est conservé ou écarté. Ce tirage à pour but d'éviter que la solution tende trop vite vers un optimum local. Le tirage dépend de la température du système. Celle-ci est donc élevée à l'initialisation de l'algorithme afin de ne pas confiner le champs d'investigation à un ensemble de solutions trop restreint. Puis elle diminue progressivement pour faire converger le système vers une solution proche de l'optimale. Si le système engendré est écarté, l'algorithme continue à évoluer avec le système d'origine. Sinon, le système engendré remplace le système d'origine et l'algorithme se poursuit.

4.5.2 Algorithme de partitionnement

Le fonctionnement de RPSA est décrit par l'algorithme 4.1. Une partition de départ P est engendrée aléatoirement en fonction de l'ensemble des tâches et du nombre de

1. Nous fixons cette borne à 1% du WCET maximal d'une tâche. Dans nos simulations, le WCET maximal est fixée à 1000 ms et une ressource est donc courte si la longueur maximale de ses sections critiques est inférieure ou égale à 10 ms.

```
   Entrées : n                                        /* Nombre de tâches */
   Entrées : m                                   /* Nombre de processeurs */
   Entrées : P =engendrerPartitionAleatoirement(n, m)
   Entrées : temp = -m / ln 0.99
   Entrées : K^max = n · m                            /* Nombre d'essais */
 1 while temp > temp^min do
 2     k = 0;
 3     while k ≠ K^max do
 4         N =engendrerVoisinAleatoirement(P);
 5         E_P =calculerEnergie(P);
 6         E_N =calculerEnergie(N);
 7         if E_N < E_P then
 8             P = N;
 9         else
10             x = (E_P − E_N) / temp;
11             if e^x ≥NombreAleatoire(0, 1) then
12                 P = N;
13             end
14         end
15         k = k + 1;
16     end
17     temp = temp / 2;
18 end
```

Algorithme 4.1: RPSA

processeurs. Une température de départ est choisie en fonction du nombre de processeurs de telle manière qu'elle soit d'autant plus haute que le nombre de processeurs est grand. Le nombre d'itérations K^{max} entre chaque refroidissement est calculé en fonction de la taille du système ($n \cdot m$). Une itération consiste à engendrer aléatoirement une partition N à partir de P. L'énergie E_P de P et E_N de N est ensuite calculée. Ce calcul est décrit par la suite dans la section 4.5.4. Si $E_N < E_P$, N remplace P. Sinon, la valeur x est calculée en fonction de l'énergie des deux partitions et de la température du système (algorithme 4.1, ligne 10). Si e^x est supérieur ou égal à un nombre tiré aléatoirement entre 0 et 1, N remplace P. Dans le cas contraire, N est écarté. L'algorithme se poursuit ainsi jusqu'à ce que la température du système ait atteint une valeur minimale.

4.5.3 Initialisation et voisinage aléatoire

La fonction `engendrerPartitionAleatoirement` place aléatoirement les n tâches de l'ensemble de tâches à partitionner sur les m processeurs. Aucune analyse d'ordonnançabilité n'est faite pour cette partition d'initialisation de RPSA. Ainsi, aucune partition de départ n'est écartée.

La fonction `engendrerVoisinAleatoirement` produit une partition N à partir d'une partition d'origine P. N est engendrée soit :

- en échangeant deux tâches. Pour cela, deux processeurs π_k et π_l non vides sont sélectionnés aléatoirement avec $\pi_k \neq \pi_l$, ainsi qu'une tâche τ_i de $\tau(\pi_k)$ et une tâche τ_j de $\tau(\pi_l)$;
- en déplaçant une tâche d'un processeur vers un autre. Pour cela, un processeur d'origine π_k non vide est sélectionné aléatoirement, ainsi qu'une tâche τ_i de $\tau(\pi_k)$ et un processeur π_l (pouvant être vide) avec $\pi_j \neq \pi_k$.

4.5.4 Calcul de l'énergie

```
1  energie = 0;
2  marge[m];
3  foreach π_j ∈ P do
4      if τ(π_j) = ∅ or π_j non ordonnançable then
5          energie = energie + 1;
6          marge[j] = 0;
7      else
8          marge[j] = Σ_{τ_i ∈ τ(π_j)} A_i
9      end
10 end
11 energie = energie + 1 / (Σ_{k=1}^{m} marge[k])
```

Algorithme 4.2: `calculerEnergie(P)`

Le fait que le *recuit simulé* puisse être appliqué sur différents problèmes d'optimisation repose sur la fonction de calcul de l'énergie du système. Cette fonction permet de donner un poids aux critères à maximiser. Dans le cas de RPSA, elle est décrite dans l'algorithme 4.2. Le critère que nous cherchons à maximiser est la robustesse temporelle du système. Notre fonction calcule donc une valeur d'énergie pour un ensemble de tâches qui est d'autant plus petite que les tâches ont une marge importante. Pour chaque processeur, deux cas de figure s'offre au sous-ensemble de tâches qui lui est affecté. Soit il est non ordonnançable, et dans ce cas, l'énergie est incrémentée de 1 pour maximiser la probabilité d'écarter cette partition. Soit la marge du processeur est calculée en faisant la somme de la marge de chacune des tâches qui lui sont assignées. L'énergie est ensuite incrémentée de l'inverse de la somme des marges de chaque processeur.

Le calcul de la marge sur le WCET ou la fréquence nécessite d'inclure le facteur de blocage des tâches. Le calcul du pire temps de blocage B_i d'une tâche τ_i induit par l'utilisation de FMLP est décrit dans l'annexe A.3.2.

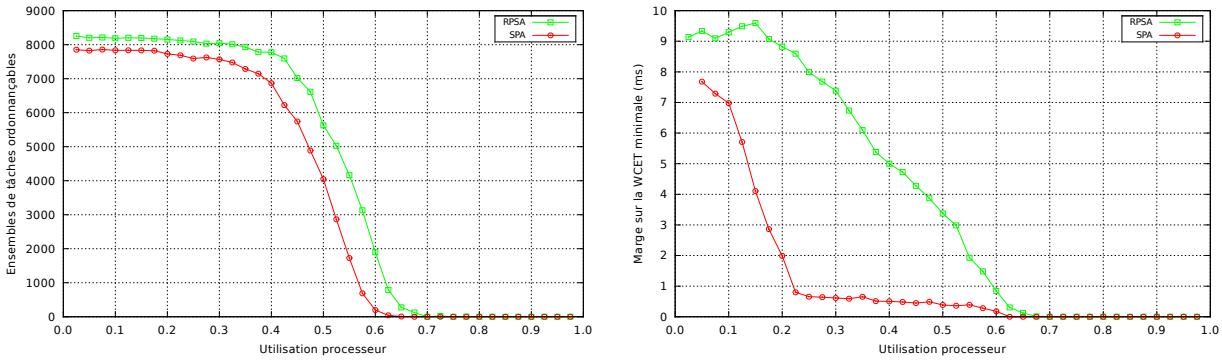
4.6 Évaluation

Dans le but de comparer notre approche basée sur le *recuit simulé* avec une approche heuristique, nous avons implanté les algorithmes RPSA et SPA dans notre simulateur. Afin de ne pas introduire de biais lié à l'utilisation de protocoles de synchronisation différents, nous avons adapté SPA pour qu'il utilise le protocole FMLP.

4.6.1 Protocole de simulation

Les résultats de simulation présentés dans les sections 4.6.2 et 4.6.3 ont été produits par notre simulateur. Chaque simulation se compose de 1000 ensembles de tâches pour chaque valeur d'utilisation parmi $[0.025, 0.05, \ldots, 0.975] \times m$ où m est le nombre de processeurs. Une simulation revient donc à l'analyse de 39000 ensembles de tâches engendrés aléatoirement. Les ensembles de tâches sont constitués de 16 tâches à échéances contraintes et la plate-forme multiprocesseur est composée de 4 processeurs. Chaque tâches utilise au plus 2 ressources (tirage aléatoire entre 0 et 2). Les durées des sections critiques sont tirées aléatoirement en fonction de WCET de la tâche.

4.6.2 Marge sur le WCET



(a) Ordonnançabilité.

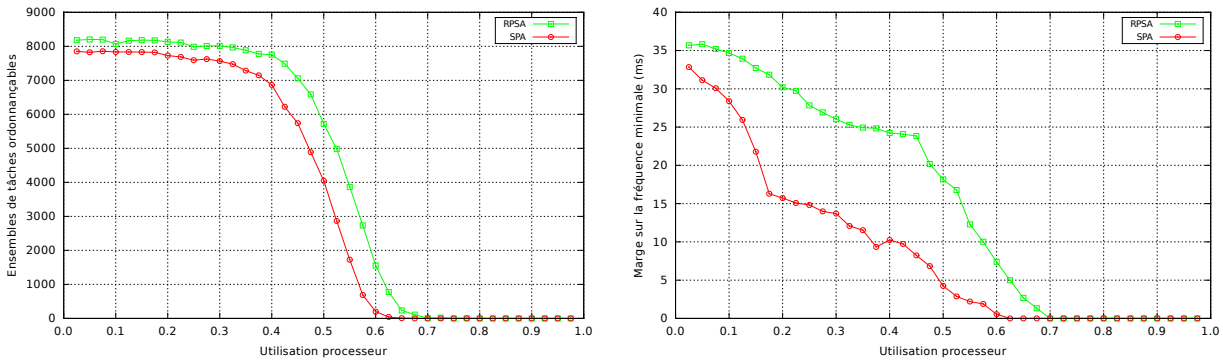
(b) Marge sur la fréquence minimale.

FIGURE 4.1 – Comparaison entre RPSA et SPA pour la marge sur le WCET.

Nous montrons dans la figure 4.1(a) que les performances en terme d'ordonnançabilité sont toujours meilleures en moyenne avec RPSA. Nous montrons également dans la figure 4.1(b) que RPSA permet de trouver en moyenne un partitionnement où la marge sur le WCET minimale est maximisée par rapport à SPA.

4.6.3 Marge sur la fréquence

De même qu'avec les tâche indépendantes, les résultats sur la marge sur la fréquence sont assez similaires à ceux sur la marge sur le WCET. Ainsi, nous montrons



(a) Ordonnançabilité.

(b) Marge sur la fréquence minimale.

FIGURE 4.2 – Comparaison entre RPSA et SPA pour la marge sur les fréquences.

dans la figure 4.2(a) que RPSA surpasse en moyenne SPA en terme d'ordonnançabilité. Nous montrons également dans la figure 4.2(b) qu'il offre de meilleures performances en terme de maximisation de la marge sur la fréquence par rapport à SPA.

4.7 Conclusion

Dans ce chapitre, nous nous sommes intéressés à la classe d'ordonnancement (FTP-FTII) dans le cas d'un ordonnancement hors-ligne. Nous avons considéré un modèle de tâches sporadiques à échéances contraintes partageant des ressources. Nous avons présenté différents protocoles de synchronisation issus de la l'état de l'art. Nous avons fait le choix d'utiliser FMLP car il tire parti des mécanismes employés par MPCP et MSRP. Nous avons proposé un algorithme de partitionnement basé sur le *recuit simulé* qui répartit un ensemble de tâches en maximisant la marge des tâches. Nous l'avons comparé par la simulation à une heuristique de partitionnement pour tâches dépendantes et nous avons montré qu'il permet de trouver des partitions où la marge sur le WCET et sur la fréquence est maximisée. Ce travail a donné lieu à la publication [FMG10b].

CHAPITRE 5

ORDONNANCEMENT À MIGRATIONS RESTREINTES

5.1 Introduction

Dans ce chapitre, nous considérons l'ordonnancement de *tâches périodiques* à *échéances contraintes*. Il s'agit d'un ordonnancement en ligne, dont la définition est donnée dans la section 1.3.1. Nous proposons un algorithme d'ordonnancement pour lequel les *migrations* sont autorisées. Si nous nous référons aux tests d'ordonnançabilité actuels, les approches d'ordonnancement par partitionnement (sans migration) sont plus performantes que les approches d'ordonnancement globales (avec migrations). En effet, Baker a proposé une comparaison empirique entre ces deux approches d'ordonnancement qui montre la supériorité de l'approche par partitionnement [Bak05]. Mais cette supériorité est imputable aux tests d'ordonnançabilité et non pas aux approches d'ordonnancement en elles-mêmes. Intuitivement, les migrations apportent une liberté supplémentaire aux tâches et augmentent leur chance de pouvoir être ordonnancées avec succès. Il convient d'ailleurs de remarquer que tous les algorithmes d'ordonnancement temps réel multiprocesseur optimaux nécessitent des migrations de tâches.

Notre algorithme appartient à la classe d'ordonnancement (FTP-FJΠ). Pour rappel, la classe d'ordonnancement FJΠ est aussi appelée classe d'ordonnancement à migrations restreintes. Un des inconvénients des algorithmes de la classe DΠ est le nombre potentiellement important de migrations qu'ils peuvent engendrer. C'est pourquoi des travaux s'intéressent à proposer des bornes sur le nombre de migrations engendrées par les algorithmes de la classe DΠ. Par exemple, l'algorithme DP-Wrap [LFPB10] n'engendre pas plus de $m - 1$ migrations par intervalle de temps (où l'intervalle de temps dépend des paramètres des tâches). Pour les algorithmes de la classe FJΠ, les migrations ne sont autorisées qu'entre deux instances d'une même tâche. Ainsi, une fois qu'une instance est démarrée sur un processeur, elle perd la possibilité de migrer. Cette approche présente l'avantage de borner le nombre de migrations à au plus une par instance.

Ha et Liu ont présenté un algorithme de la classe (FTP-FJΠ) [HL94]. Ils ont également prouvé que cet algorithme n'était pas prédictible. Nous en analyserons les raisons puis nous présenterons la contribution majeure de ce chapitre : un algorithme prédictible pour la classe (FTP-FJΠ).

Ce chapitre est organisé de la manière suivante. Dans la section 5.2, nous présentons le modèle considéré, ainsi que les notations employées. Dans la section 5.3, nous proposons un nouvel algorithme d'ordonnancement basé sur l'approche d'ordonnancement FJΠ (aussi appelée ordonnancement à migrations restreintes). Bien qu'étant de la classe FTP, cet algorithme utilise le paramètre de laxité pour prendre les décisions d'ordonnancement. Mais contrairement à LLF, il ne n'utilise pas la laxité pour attribuer une priorité, mais pour choisir le processeur sur lequel activer l'instance. Dans la section 5.4, nous étudions la propriété de viabilité de notre algorithme. Dans la section 5.5, nous proposons deux tests d'ordonnançabilité. Le premier est un test nécessaire et suffisant. Le second est un test suffisant, plus simple à calculer. Finalement, nous faisons la synthèse de ces résultats dans la section 5.6.

5.2 Terminologie

Notation	Définition
τ_i	La tâche d'indice i
J_i	Une instance de la tâche τ_i
$J_{i,k}$	La $k^{\text{ième}}$ instance de la tâche τ_i
$r_{i,k}$	L'instant d'activation de l'instance $J_{i,k}$
$e_{i,k}$	Le coût d'exécution de l'instance $J_{i,k}$
$e^*_{i,k}(t)$	Le coût d'exécution restant de l'instance $J_{i,k}$ à l'instant t
$d_{i,k}$	L'échéance absolue de l'instance $J_{i,k}$
$L_{i,k}(t, \pi_j)$	La laxité de $J_{i,k}$ sur le processeur π_j à l'instant t
π_j	Le processeur d'indice j
$hp(\pi_j, J_{i,k})$	L'ensemble des instances plus prioritaires que $J_{i,k}$ sur le processeur π_j
$lp(\pi_j, J_{i,k})$	L'ensemble des instances moins prioritaires que $J_{i,k}$ sur le processeur π_j
$J(\pi_j)$	L'ensemble des instances ordonnancées sur le processeur π_j
C_i	Le WCET de τ_i
D_i	L'échéance relative de τ_i
T_i	La période de τ_i
$D_{max}(k)$	L'échéance relative maximale parmi $\{D_1, \ldots, D_k\}$
δ_i	La densité de τ_i ($\delta_i = C_i / \min(D_i, T_i)$)
$\delta_{min}(k)$	La densité minimale parmi les densités des tâches $\{\tau_1, \ldots, \tau_k\}$

TABLE 5.1 – Notations employées dans le chapitre 5.

Nous rappelons que $r_{i,k}$ (respectivement $e_{i,k}$, $e^*_{i,k}(t)$, $d_{i,k}$ et $L_{i,k}(t)$) peut être notée r_i (respectivement e_i, $e^*_i(t)$, d_i et $L_i(t)$) lorsque nous ne faisons pas référence à une instance particulière de la tâche τ_i.

Définition 5.1 (Laxité). *La laxité $L_{i,k}(t, \pi_j)$ d'une instance $J_{i,k}$ sur le processeur π_j à l'instant t est l'intervalle de temps entre la terminaison de $J_{i,k}$ et son échéance absolue. $L_{i,k}(t, \pi_j)$ est définie sur l'intervalle $[r_{i,k}, d_{i,k}]$.*

5.3 Algorithme d'ordonnancement

5.3.1 Travaux existants

À notre connaissance, l'approche FJII a été peu étudiée. Nous pouvons citer les travaux de Baruah et Carpenter. Ils ont proposé deux algorithmes dans [BC03], *r-EDF* basé sur EDF et *r-PriD* basé sur *PriD* [GFB03]. Ce travail a été étendu au modèle des architectures à processeurs uniformes par Funk et Baruah [FB05]. Un algorithme à migrations restreintes fondé sur EDF a été proposé par Anderson, Bud et Devi [ABD08], mais dans le cas d'un ordonnancement temps réel souple. Cette approche a aussi été utilisée par Dorin *et al.* pour proposer un algorithme de partitionnement des instances hors-ligne et un ordonnancement en ligne basé sur EDF [DMYGR10]. Tous ces travaux considèrent que l'algorithme d'attribution des priorités est EDF. Ha et Liu ont présenté une manière d'ordonnancer des instances avec un algorithme de la classe (*FixedTaskPriority-FixedJobProcessor*) [HL94]. Ils ont montré que cette approche d'ordonnancement n'était pas prédictible. Fisher a plus récemment proposé un une condition suffisante d'ordonnancement pour cet algorithme [Fis07]. Mais il n'a pas abordé la notion de prédictibilité.

5.3.2 Laxité

Lorsque l'on parle d'ordonnancement avec laxité, il est tout naturel de penser à l'algorithme LLF [Leu89, DM89]. Nous considérons toutefois dans ce chapitre un algorithme de la classe d'ordonnancement FTP. La laxité est utilisée pour prendre des décisions d'ordonnancement aux instants d'activations des instances, c'est-à-dire à chaque début d'instance, et pas pendant son exécution comme avec LLF. Nous considérons le critère de laxité car nous nous intéressons à la robustesse, qui consiste à exploiter le temps pendant lequel le processeur est oisif pour permettre des déviations sur les paramètres temporels. Même si la valeur de laxité calculée à l'instant de l'activation de l'instance peut être amenée à diminuer au cours de son exécution, la connaissance de cette valeur peut permettre la mise en place d'un mécanisme similaire au *vol de temps creux*. Le vol de temps creux consiste à récupérer les temps d'oisiveté afin d'exécuter entre autres des traitements apériodiques.

À l'instant de son activation $t = r_{i,k}$, la laxité $L_{i,k}(t)$ de l'instance $J_{i,k}$ sur le proces-

instance	r_i	d_i	$[e_i^-, e_i^+]$
J_1	0	10	$[5, 5]$
J_2	0	10	$[2, 6]$
J_3	4	15	$[8, 8]$
J_4	0	20	$[10, 10]$
J_5	5	200	$[100, 100]$
J_6	7	25	$[2, 2]$

TABLE 5.2 – Paramètres temporels des instances de l'exemple de la figure 5.1.

seur π_j est donnée par :

$$L_{i,k}(t) = D_i - C_i - \sum_{J_j \in hp(\pi_j, J_{i,k})} e_j^*(t) \tag{5.1}$$

Les valeurs données par l'équation 5.1 correspondent à la laxité dans le pire cas. La laxité $L_{i,k}(t)$ d'une instance $J_{i,k}$ est une fonction décroissante qui décroît lorsqu'une instance de plus haute priorité que $J_{i,k}$ est activée sur le même processeur.

5.3.3 Anomalies d'ordonnancement

La preuve de non prédictibilité de Ha et Liu est basée sur l'exemple donné par la figure 5.1. Leur algorithme d'ordonnancement fonctionne de la manière suivante. Lorsqu'une instance est activée, elle peut être soit démarrée sur un processeur libre, soit mise en attente de libération d'un processeur. Un processeur est considéré libre relativement au niveau de priorité de l'instance active si ce processeur n'exécute pas d'instance de priorité supérieure. Si la tâche nouvellement activée préempte une instance en cours d'exécution, cette dernière est alors mise en attente de libération du processeur sur lequel elle s'exécutait.

Les paramètres temporels des 6 instances sont donnés dans la table 5.2. Il faut remarquer que l'instance J_2 peut être exécutée pour une durée comprise entre 2 et 6 unités de temps. Toutes les autres instances ne peuvent être exécutées que pour une durée fixée ($e_i^- = e_i^+$). Dans la figure 5.1(a), l'instance J_2 est exécutée pour une durée $e_2 = 6$, soit $\forall i \in [1 - 6]$, $e_i = e_i^+$. Dans la figure 5.1(b), $e_2 = 2$ et $\forall i \in [1 - 6]$, $e_i = e_i^-$. Ces deux figures représentent donc l'ordonnancement dans le pire cas (figure 5.1(a)) et l'ordonnancement dans le meilleur cas (figure 5.1(b)) relativement aux coûts d'exécution des instances. Pour ces deux ordonnancements, aucune échéance n'est dépassée. Pourtant, dans la figure 5.1(c), l'instance J_4 rate son échéance alors que l'instance J_2 est exécutée pour une durée $e_2 = 3$. Nous pouvons remarquer que ce phénomène se produit car le processeur π_2 devient disponible à l'instant $t = 4$ car J_4 est moins prioritaire que J_3. J_3 est alors admise sur π_2, mais la laxité de J_4 n'est plus suffisante pour lui permettre de terminer son exécution avant son échéance. Nous pouvons aussi remarquer que le meilleur cas pour J_4 se produit lorsque J_2 est exécutée pour $e_2 = 5$ (figure 5.1(d)). Cet

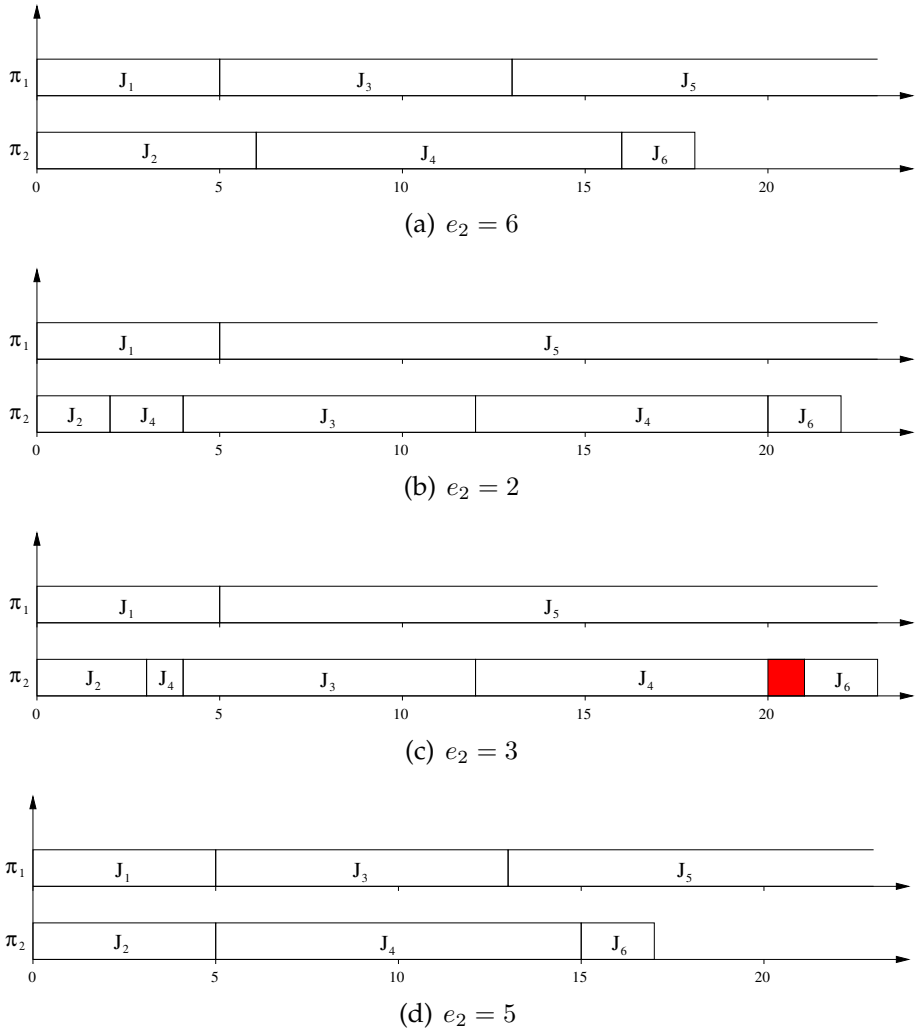


FIGURE 5.1 – Exemple de la non prédictibilité d'un algorithme d'ordonnancement à migrations restreintes [HL94].

algorithme n'est donc pas prédictible.

5.3.4 Description de l'algorithme

Nous proposons l'algorithme *r-SP_wl* (*r-SP* faisant référence à *r-EDF* mais avec un ordonnancement FTP, aussi appelé *Static-Priority* (SP) dans l'état-de-l'art, et *wl* pour *with laxity*). Cet algorithme prend ses décisions d'ordonnancement lors de l'activation d'une instance en fonction de la laxité des instances actives.

L'ordonnanceur doit stocker pour chaque instance ordonnancée sa valeur de laxité pour pouvoir la consulter et la mettre à jour. Nous considérons des tâches à échéances contraintes, ce qui implique qu'au plus une seule instance d'une même tâche peut être activée à l'instant t. Nous faisons l'hypothèse d'un ordonnanceur capable d'attribuer une priorité différente à chaque tâche. Nous utilisons alors une table d'association (à hachage parfait) pour associer une valeur de laxité à un niveau de priorité. En effet, une telle structure d'association permet d'obtenir la valeur associée en un temps constant.

Les instances sont admises sur un processeur à l'instant de leur activation. Une instance $J_{i,k}$ peut être admise sur un processeur π_j à l'instant $t = r_{i,k}$ si et seulement si :

- la laxité de $J_{i,k}$ est supérieure ou égale à 0. $L_{i,k}(t) \geq 0$ avec $L_{i,k}(t)$ donnée par l'équation (5.1) ;
- la laxité des instances de priorité inférieure à $J_{i,k}$ est supérieure ou égale à 0 après admission de $J_{i,k}$:

$$\forall J_j \in lp(\pi_j, J_{i,k}),\ L_j(t) - e_{i,k} \geq 0 \tag{5.2}$$

La condition donnée par l'équation (5.2) garantit qu'aucune instance de priorité inférieure à celle de l'instance $J_{i,k}$ nouvellement admise ne dépassera son échéance. Nous noterons que cette phase d'admission consiste à garantir, qu'à l'instant t, l'instance ayant la plus petite valeur de laxité, parmi toutes les instances de priorité inférieure, en aura suffisamment pour ne pas dépasser son échéance. Cette opération consiste à vérifier la valeur de laxité de toutes les instances de priorité inférieure à celle de $J_{i,k}$. Pour un processeur π_j, elle a une complexité linéaire en le nombre de tâches.

Lorsqu'une instance $J_{i,k}$ est activée ou admise, une entrée est ajoutée dans la structure de données pour stocker sa valeur de laxité telle que calculée par l'équation (5.1). Avant de calculer cette valeur, le coût d'exécution restant $e_j(r_{i,k})^*$ de toutes les instances J_j plus prioritaires que $J_{i,k}$ doit être mis à jour. Le coût d'exécution restant des instances en cours d'exécution est décrémenté de la durée écoulée depuis le dernier instant d'ordonnancement.

Quand plusieurs processeurs sont disponibles lors de l'admission d'une instance, notre algorithme admet celle-ci sur le processeur pour lequel la valeur de laxité minimale est la plus grande. La laxité minimale à l'instant t d'un processeur π_j est donnée par $\min_{J_i \in J(\pi_j)} L_i(t)$. Ce choix n'est pas optimal. Il permet néanmoins une bonne répartition des instances sur les différents processeurs dans le cas où le système n'est pas trop chargé. Cela a pour effet de pouvoir offrir plus de marge aux tâches. Notre algorithme d'ordonnancement étant un algorithme en ligne avec des migrations, il est difficile de calculer cette valeur de marge autrement qu'en déroulant l'ordonnancement sur une période d'étude.

À la terminaison d'une instance, celle-ci est arrêtée et les ressources qu'elle a utilisées sont libérées. Il est important de noter que, d'après l'équation (5.1), la laxité d'une instance est calculée à partir de son WCET. Si cette instance J_i termine son exécution avant son WCET, l'ordonnanceur continue de considérer l'interférence de celle-ci sur les instances moins prioritaires jusqu'à ce que son WCET ait été virtuellement consommé. Il adoptera donc un comportement oisif en n'admettant pas d'instance supplémentaire qui ne l'aurait pas été dans le cas où J_i aurait été exécutée pour une durée égale à son WCET.

Dans l'algorithme 5.1, nous décrivons le fonctionnement de *r-SP_wl*. Tout d'abord, nous considérons que l'ordonnanceur est réveillé par des évènements (signaux, interruptions) correspondant à l'activation ou à la terminaison d'une instance. La donnée

```
   Entrées : Π
   Entrées : τ
   Données : Q_g
   Données : Q_l
 1 pour π_j ∈ Π faire                                  /* Initialisation */
 2 |  Q_l(π_j) ← ∅;
 3 fin
 4 Q_g ← ∅;
 5 tant que évènement E reçu pour J_i faire            /* Ordonnancement */
 6 |  si E = activation de J_i alors
 7 |  |  pour π_j ∈ trierParLaxiteDecroissante(Π) faire
 8 |  |  |  si L_i(t_E) ≥ 0 et min(lp(π_j, J_i)) >= e_i alors
 9 |  |  |  |  empiler J_i dans Q_l(π_j);
10 |  |  |  |  saut à l'instruction 16;
11 |  |  |  fin
12 |  |  fin
13 |  |  empiler J_i dans Q_g;
14 |  sinon si E = terminaison de J_i alors
15 |  |  arrêter J_i;
16 |  fin
17 |  J_j ← instance la plus prioritaire de Q_g ∪ Q(π(J_i));
18 |  J_k ← instance la plus prioritaire de π(J_i);
19 |  si J_j plus prioritaire que J_k alors
20 |  |  dépiler J_j;
21 |  |  démarrer J_j;
22 |  |  empiler J_k dans Q_l(π(J_k));
23 |  fin
24 fin
```

Algorithme 5.1: Algorithme d'ordonnancement *r-SP_wl*.

Q_g représente la file globale des instances actives. C'est dans cette file que sont stockées les instances qui n'ont pas pu être assignées sur un processeur au moment de leur activation en attente de libération d'un processeur. La donnée Q_l représente l'ensemble des files locales aux processeurs des instances actives. Ainsi, $Q_l(\pi_j)$ représente la file locale au processeur π_j. C'est dans cette file qu'est stockée une instance, déjà assignée sur un processeur, qui est préemptée par une instance de plus haute priorité.

La boucle événementielle (lignes [5 - 24]) traite les activations et terminaisons d'instances. Lorsqu'une instance J_i est activée (ligne 6), l'ensemble des processeurs trié par laxité décroissante est parcouru (ligne 7). Si la laxité est suffisante pour admettre l'instance sur un processeur π_j (ligne 8), J_i est ajoutée à la file $Q_l(\pi_j)$. Le traitement de l'évènement est alors terminé. Si aucun processeur n'est disponible (ligne 13), J_1 est ajoutée à la file globale Q_g.

À la ligne 7, nous omettons volontairement la description de la gestion des processeurs en utilisant la fonction `trierParLaxiteDecroissante`. Cette gestion peut

être opérée par le maintien d'un arbre AVL pour que la mise à jour de l'ensemble des processeurs soit faite avec une complexité en $O(log_2(m))$. Un arbre AVL (pour *Adelson-Velsky et Landis*) est un arbre binaire de recherche automatiquement équilibré.

Lorsque J_i termine son exécution (ligne 14), elle est arrêtée et l'ordonnanceur supprime les informations concernant sa laxité et son coût d'exécution restant.

Quand le traitement des évènements est terminé, l'ordonnancement doit mettre à jour l'instance active sur le processeur π_j (ligne [17 - 23]). J_j est l'instance la plus prioritaire parmi la file globale Q_g et la file locale $Q_l(\pi_j)$ de π_j. J_k est l'instance la plus prioritaire de π_j. Si J_j est plus prioritaire que J_k, elle la préempte et démarre son exécution sur π_j. J_k est alors ajoutée à la file $Q_l(\pi_j)$. S'il n'y a pas d'instance J_j, aucune mise à jour n'est nécessaire. S'il n'y a pas d'instance J_k, J_j est démarrée sans préemption.

5.4 Viabilité

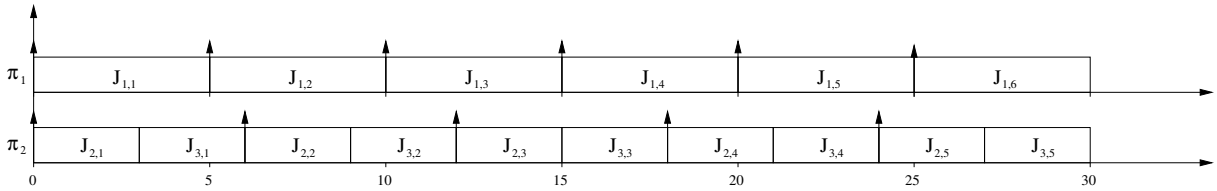
La viabilité d'un algorithme d'ordonnancement, ou tout du moins la viabilité du test d'ordonnançabilité qui lui est associé, est une propriété importante. Elle apporte notamment la garantie que l'algorithme puisse être implanté sans que des anomalies d'ordonnancement ne se produisent. Nous montrons dans cette section que *r-SP_wl* n'est pas viable au sens strict de la définition donnée dans le chapitre 2. Nous montrons qu'il l'est dans le cas des tâches périodiques et non pas sporadiques.

5.4.1 Résistance aux diminutions de coûts d'exécution

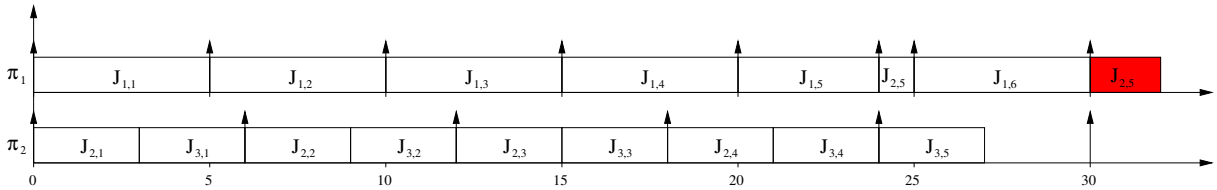
L'anomalie qui se produit avec l'algorithme (FTP-FJΠ) de Ha et Liu est imputable à la définition de la disponibilité processeur qui lui est associée. En effet, avec cet algorithme, un processeur est disponible pour une instance J_i si il est soit inactif, soit en train d'exécuter une instance de priorité inférieure à celle de J_i. Dans le cas de *r-SP_wl*, un processeur π_j est disponible pour l'admission d'une instance J_i si et seulement si la laxité des instances déjà assignées sur π_j reste supérieure ou égale à 0 après admission de J_i.

Comme l'ont montré Ha et Liu, le problème de prédictibilité de la classe d'ordonnancement (FTP-FJΠ) vient du fait qu'une diminution de coût d'exécution peut permettre l'admission d'une tâche qui aurait été assignée sur un processeur différent. Cette instance peut alors interférer sur l'exécution d'instances de priorités inférieures et leur faire rater leur échéance. Comme nous le montrons dans la figure 5.2, *r-SP_wl* souffrirait d'anomalies si son comportement ne faisait pas de lui un algorithme d'ordonnancement oisif.

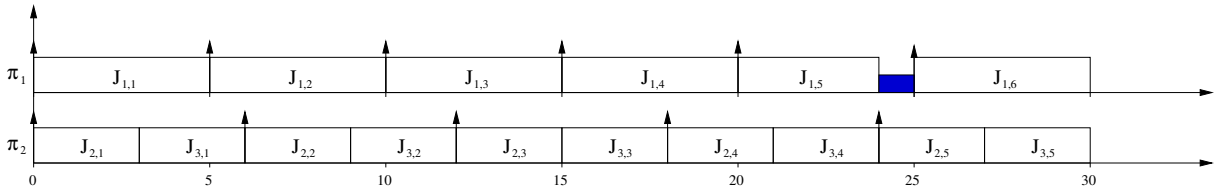
Dans la figure 5.2(a), nous représentons un ensemble d'instances ordonnançables. L'ensemble de tâches qui les engendre est caractérisé par $\{\tau_1(O_1 = 0, C_1 = 5, D_1 = 5, T_1 = 5), \tau_2(O_2 = 0, C_2 = 3, D_2 = 6, T_2 = 6), \tau_3(O_3 = 0, C_3 = 3, D_3 = 6, T_3 = 6)\}$.



(a) Ensemble d'instances ordonnançable.



(b) Ensemble d'instances non ordonnançables sans oisiveté.



(c) Ensemble d'instances ordonnançables avec oisiveté.

FIGURE 5.2 – Exemple de non prédictibilité en migrations restreintes avec et sans oisiveté.

Dans la figure 5.2(b), nous représentons ce même ensemble où l'instance $J_{1,5}$ a un coût d'exécution $e_{1,5} = 4$. À l'instant $t = 24$, les processeurs π_1 et π_2 sont donc tous deux disponibles. L'instance $J_{2,5}$, respectivement $J_{3,5}$, est donc assignée et démarrée sur le processeur π_1, respectivement π_2. Lorsqu'à l'instant $t = 25$, l'instance $J_{1,6}$ est activée, elle ne peut être assignée sur aucun processeur sans causer le dépassement d'une échéance. Dans la figure 5.2(c), nous représentons le l'ordonnancement de cet ensemble d'instances réalisé par *r-SP_wl*. Mais cette fois-ci, le processeur π_1 reste oisif entre les instants $t = 24$ et $t = 25$.

Proposition 5.1. r-SP_wl *est un algorithme d'ordonnancement oisif résistant aux diminutions de coûts d'exécution.*

Démonstration. La démonstration est faite par récurrence sur la priorité des instances. Soit J un ensemble de n instances. Nous notons J_i^- l'instance J_i dont le coût d'exécution e_i^- est tel que $e_i^- \leq e_i$. Nous notons aussi $L_i^-(t, \pi_j) = D_i - C_i - \sum_{J_h^- \in hp(\pi_j, J_i^-)} e_j^{-*}(t)$ la laxité de J_i^- à l'instant t sur le processeur π_j. L'hypothèse de récurrence est $\forall\ 1 \leq i \leq n,\ L_i^-(t, \pi_j) \geq L_i(t, \pi_j)$. L'hypothèse est vraie au rang 1 car J_1 étant l'instance la plus prioritaire, $L_i^-(t, \pi_j) = L_i(t, \pi_j)$ par définition de la laxité. Supposons maintenant que la récurrence soit vraie pour les i premières instances. Nous vérifions maintenant que

la relation $L^-_{i+1}(t, \pi_j) \geq L_{i+1}(t, \pi_j)$ est vraie pour l'instance J_{i+1}. Par contradiction :

$$L^-_{i+1}(t, \pi_j) < L_{i+1}(t, \pi_j) \tag{5.3}$$

$$\Rightarrow \sum_{J^-_h \in hp(\pi_j, J^-_i)} e^{-*}_j(t) > \sum_{J_h \in hp(\pi_j, J_i)} e^*_j(t) \tag{5.4}$$

Comme π_j est resté oisif lorsque des instances plus prioritaires que J^-_{i+1} ont terminé leur exécution avant leur WCET, nous avons $\sum_{J^-_h \in hp(\pi_j, J^-_i)} e^{-*}_j(t) = \sum_{J_h \in hp(\pi_j, J_i)} e^*_j(t)$, ce qui est en contradiction avec l'équation 5.4. L'hypothèse de récurrence est donc vérifiée. □

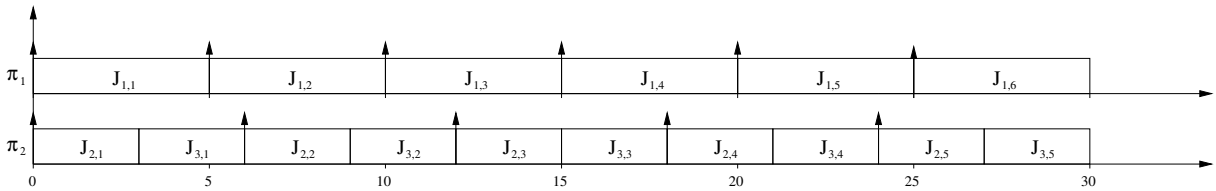
5.4.2 Résistance aux augmentations d'échéances

Une autre propriété de l'algorithme *r-SP_wl* est d'être résistant aux augmentations d'échéance. Ce type d'anomalie est lié à un problème d'implantation de l'algorithme. En effet, l'analyse d'ordonnançabilité est en général faite en considérant un ensemble de tâches avec des paramètres temporels fixés. Si certaines instances ont des échéances plus grandes et que l'ordonnanceur les considère (comme l'algorithme EDF), alors il peut en résulter des anomalies d'ordonnancement. Il suffit alors de prendre l'échéance fixe de la tâche comme critère d'ordonnancement pour éviter toute anomalie de ce type.

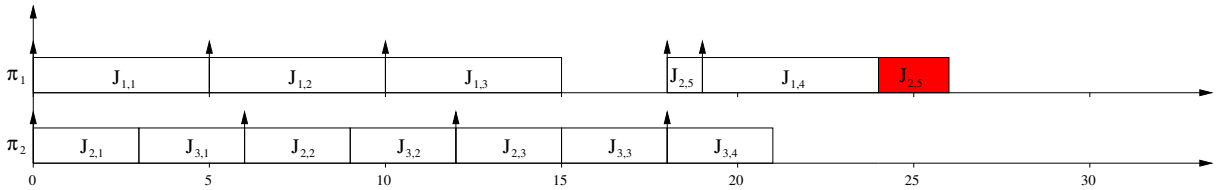
Proposition 5.2. r-SP_wl *est résistant aux augmentations d'échéances.*

Démonstration. Il suffit de remarquer que la laxité des instances ne dépend pas de leur échéance absolue mais de l'échéance relative de la tâche qui les a engendrées. □

5.4.3 Résistance aux augmentations de périodes



(a) Ensemble d'instances ordonnançable.



(b) Ensemble d'instances non ordonnançable.

FIGURE 5.3 – Exemple de non résistance aux augmentations de périodes.

Dans la figure 5.3(a), nous représentons un ensemble d'instances engendrées par l'ensemble de tâches $\{\tau_1(O_1 = 0, C_1 = 5, D_1 = 5, T_1 = 5), \tau_2(O_2 = 0, C_2 = 3, D_2 = 6, T_2 = 6), \tau_3(O_3 = 0, C_3 = 3, D_3 = 6, T_3 = 6)\}$. Cet ensemble est ordonnançable. Dans la figure 5.3(b), nous représentons le même ensemble d'instance à la différence que cette fois-ci, la période d'inter-arrivée entre les instances $J_{1,3}$ et $J_{1,4}$ est de 9 unités de temps au lieu de 5. Il en résulte que l'ensemble n'est plus ordonnançable.

Proposition 5.3. r-SP_wl *n'est pas résistant aux augmentations de périodes.*

Démonstration. La démonstration est donnée par le contre-exemple de la figure 5.3. □

Proposition 5.4. r-SP_wl *n'est pas viable.*

Démonstration. *r-SP_wl* est résistant aux diminutions de coûts d'exécution d'après la propriété 5.1, aux augmentations d'échéances d'après la propriété 5.2, mais pas aux augmentations de périodes d'après la propriété 5.3. *r-SP_wl* n'est donc viable par définition de la viabilité. □

Ce résultat peut sembler négatif. Toutefois, *r-SP_wl* étant prédictible, ce résultat signifie que notre algorithme ne permet pas d'ordonnancer des *tâches sporadiques*. Traiter le cas du modèle sporadique est donc une perspective intéressante pour la poursuite de cette étude.

5.5 Tests d'ordonnançabilité

5.5.1 Test nécessaire et suffisant

L'algorithme *r-SP_wl* étant prédictible, nous pouvons définir un intervalle d'étude pour pouvoir décider de l'ordonnançabilité d'un ensemble de tâches τ.

Définition 5.2 (Intervalle de faisabilité)**.** *Soit un ensemble de tâches τ à ordonnancer sur un ensemble de processeurs Π. Un intervalle de faisabilité est un intervalle fini tel que si aucune échéance n'est dépassée pour les instances activées dans cet intervalle, alors l'ensemble de tâches est ordonnançable.*

Un scénario d'activation asynchrone pour un ensemble τ de n tâches périodiques est l'ensemble $\{O_1, \ldots, O_n\}$ tel que $\exists i, 1 \leq i \leq n, \quad O_i \neq 0$. Dans [CG06], Cucu et Goossens ont donné un intervalle de faisabilité pour un ensemble de tâches avec scénario d'activations asynchrone. Cet intervalle considère que l'ordonnancement d'un ensemble de tâches avec scénario asynchrone est périodique à partir de l'instant S_n. La période P est égale au plus petit commun multiple des périodes des tâches de τ. S_n correspond à l'instant où toutes les tâches sont activées et est défini récursivement par :

- $S_1 = O_1$;

– $S_i = \max(O_i, O_i + \left\lceil \frac{S_{i-1}-O_i}{T_i} \right\rceil T_i)$.

Théorème 5.1 (Théorème 9 [CG06]). *Pour tout algorithme FTP préemptif $\mathcal{A}$, si un ensemble de tâches asynchrones à échéances contraintes τ est ordonnançable avec $\mathcal{A}$, alors l'ordonnancement de τ produit par $\mathcal{A}$ sur m processeurs est périodique de période P à partir de S_n.*

L'ordonnancement produit par *r-SP_wl* étant périodique, il est donc nécessaire de définir un intervalle de faisabilité pour pouvoir vérifier le respect des échéances.

Théorème 5.2 (Théorème 12 [CG06]). *Pour tout ensemble τ de tâches ordonnançable avec m processeurs, $[X_1, S_n + P]$ est un intervalle de faisabilité.*

L'intervalle commence à l'instant X_1 correspondant à l'instant où ont été activées les instances qui se terminent après S_n. X_1 est défini récursivement par :
– $X_n = S_n$;
– $X_i = O_i + \left\lfloor \frac{X_{i+1}-O_i}{T_i} \right\rfloor T_i$.

Proposition 5.5 (Condition nécessaire et suffisante). *L'étude de l'ordonnancement produit par* r-SP_wl *sur l'intervalle $[X_1, S_n + P]$ est une condition nécessaire et suffisante d'ordonnançabilité.*

Démonstration. L'ordonnancement produit par *r-SP_wl* est prédictible (proposition 5.1) et périodique (théorème 5.1). La démonstration découle du théorème 5.2. □

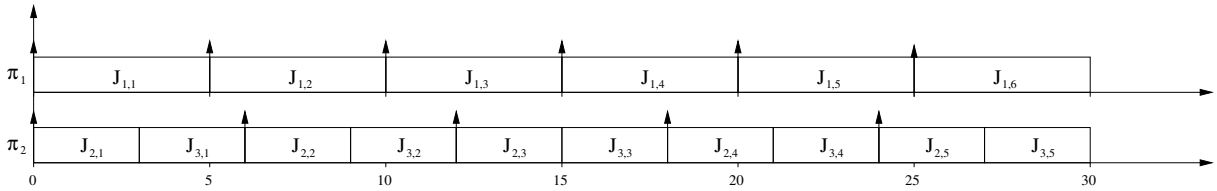
Pour pouvoir décider de l'ordonnançabilité d'un ensemble τ de tâches sur un ensemble Π de processeurs, nous devons identifier le pire scénario d'activations.

Définition 5.3 (Pire scénario d'activations). *Un scénario d'activations caractérisé par $\{O_1, \ldots, O_n\}$ pour un ensemble τ de n tâches est le pire scénario d'activations si l'ordonnançabilité (respectivement la non ordonnançabilité) de τ avec ce scénario implique l'ordonnançabilité (respectivement la non ordonnançabilité) de τ pour tout scénario.*

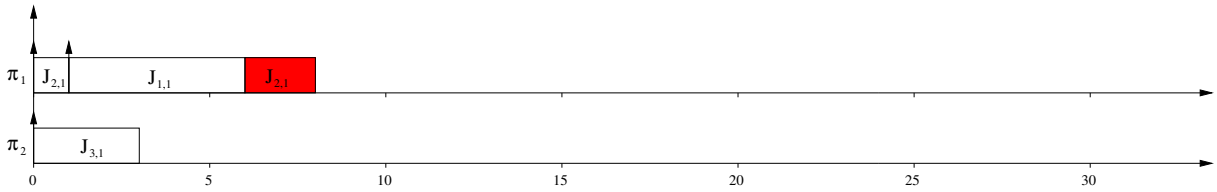
Dans le cas d'un ordonnancement monoprocesseur en priorités fixes, le pire scénario d'activations est le scénario synchrone. Le scénario d'activations synchrone correspond au scénario $O_i = 0$ pour $1 \leq i \leq n$.

Dans la figure 5.4, nous représentons un ensemble d'instances engendré par l'ensemble de tâches $\{\tau_1(O_1 = 0, C_1 = 5, D_1 = 5, T_1 = 5), \tau_2(O_2 = 0, C_2 = 3, D_2 = 6, T_2 = 6), \tau_3(O_3 = 0, C_3 = 3, D_3 = 6, T_3 = 6)\}$. Cet ensemble est ordonnançable dans le cas du scénario d'activations synchrones comme nous le représentons dans la figure 5.4(a). Dans la figure 5.4(b), nous représentons l'ordonnancement produit par l'ensemble de tâches $\{\tau'_1, \tau_2, \tau_3\}$ où τ'_1 est caractérisée par $(O'_1 = 1, C'_1 = 5, D'_1 = 5, T'_1 = 5)$. L'instance $J_{2,1}$ dépasse son échéance et cet ensemble de tâches n'est donc par ordonnançable avec *r-SP_wl*.

Proposition 5.6. *Pour l'algorithme* r-SP_wl, *le pire scénario d'activations pour un ensemble τ de tâches à ordonnancer sur un ensemble Π de processeurs n'est pas le scénario synchrone.*



(a) Ensemble d'instances ordonnançable avec un scénario synchrone.



(b) Ensemble d'instances non ordonnançable avec un scénario asynchrone.

FIGURE 5.4 – Exemple de scénario synchrone et asynchrone.

Démonstration. La démonstration est donnée par le contre-exemple de la figure 5.4. □

Pour pouvoir décider de l'ordonnançabilité d'un ensemble τ de tâches sur un ensemble Π de m processeurs, il faut donc considérer tous les scénarios d'activations possibles pour identifier le pire. Goossens a montré qu'il existait $\frac{\prod_{i=1}^{n} T_i}{P}$ scénarios non équivalents [GD97]. Avec l'algorithme *r-SP_wl*, il faut étudier tous les scénarios d'activations non équivalents sur l'intervalle $[X_1, S_n + P]$.

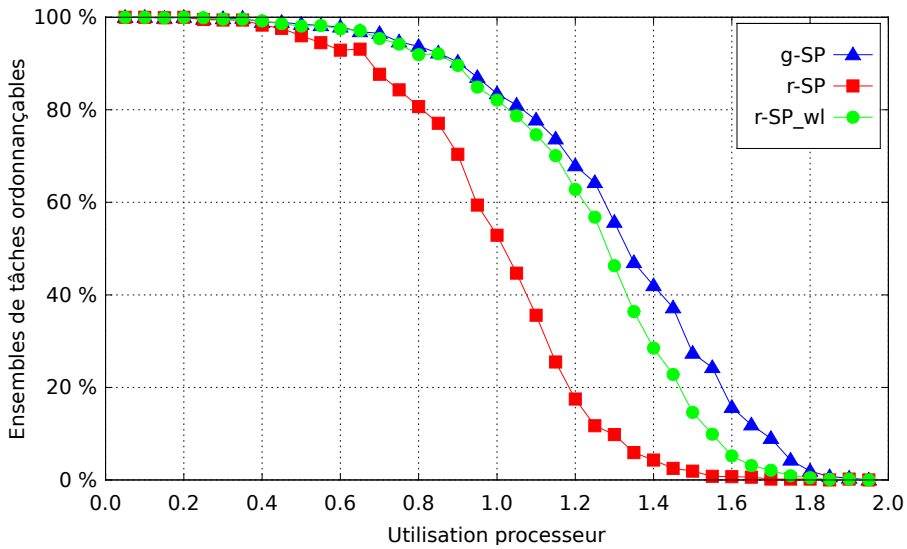


FIGURE 5.5 – Comparaison de l'ordonnançabilité d'algorithmes FTP : algorithme global, algorithme à migrations restreintes et r-SP_wl.

Dans la figure 5.5, nous comparons l'ordonnançabilité de trois algorithmes avec une condition nécessaire et suffisante. Nous simulons l'ordonnancement d'un ensemble de tâches sur un intervalle d'étude et nous vérifions qu'aucune échéance n'est dépassée. Cette approche requiert un temps de calcul important[1], c'est pourquoi nous avons

1. La simulation décrite dans ce chapitre a duré environ une semaine sur une machine octo-cœur.

limité la simulation à 1000 ensembles de tâches périodiques à échéances contraintes pour chaque valeur d'utilisation comprise dans l'intervalle $[0,025, 0,05, \ldots, 0,975]$ (soit 39000 ensembles de tâches au total). De plus, la longueur de l'intervalle d'étude étant exponentielle en le nombre de tâches, nous avons limité le nombre de tâches à 6 par ensemble de tâches. L'ordonnancement de ces ensembles de tâches est simulé sur 2 processeurs.

L'algorithme d'ordonnancement identifié par *g-SP* est l'algorithme global FTP. Celui que nous désignons par *r-SP* est l'algorithme (FTP-FJII) présenté par Ha et Liu. Hormis le fait que les migrations d'instances soient autorisées pour *g-SP* mais pas pour *r-SP*, ces deux algorithmes ont été implantés de manière relativement similaire. Dès qu'une instance devient active, elle est démarrée sur le premier processeur disponible. Sinon, elle est mise en attente dans une file globale. Un processeur est disponible pour une instance $J_{i,k}$ lorsqu'il n'exécute pas d'instance plus prioritaire que $J_{i,k}$. Les tâches étant à échéances contraintes, l'algorithme DM est utilisé pour attribuer les priorités des tâches.

Nous remarquons que l'algorithme *g-SP* offre les meilleurs résultats en terme d'ordonnançabilité. Mais nous remarquons aussi que la distance entre la courbe représentant l'algorithme *r-SP* et celle représentant notre algorithme *r-SP_wl* est beaucoup plus grande que la distance entre les courbes de *r-SP_wl* et *g-SP*. Nous obtenons ainsi une ordonnançabilité presque aussi bonne avec *r-SP_wl* qu'avec un ordonnancement global, mais en garantissant au plus une migration par instance. Nous pouvons observer que l'utilisation de la laxité du système permet d'améliorer les décisions d'ordonnancement en moyenne. Dans le but de ne pas biaiser la simulation, nous avons implanté l'algorithme *r-SP* pour qu'il choisisse les processeurs inactifs pour admettre les instances plutôt que de préempter des instances de priorités inférieures.

5.5.2 Test suffisant

Le test nécessaire et suffisant donné dans la section 5.5.1 étant complexe à calculer, nous proposons un test suffisant d'ordonnançabilité basé sur l'utilisation de l'ensemble de tâches à ordonnancer. Ce test d'ordonnançabilité s'appuie sur la charge du système, et plus particulièrement sur la définition de *LOAD*. La démonstration de cette borne s'inspire de celle décrite dans [BF08].

Définition 5.4 (DBF). *Pour un intervalle de longueur t, la fonction $DBF(\tau_i, t)$ d'une tâche sporadique τ_i borne le temps d'exécution des instances de τ_i qui sont activées et ont leur échéance dans cet intervalle.*

Il a été montré dans [BMR90] que $DBF(\tau_i, t) = \max\left(0, (\left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1)C_i\right)$.

Définition 5.5 (LOAD). *Pour tout k, le paramètre* LOAD *est défini de la manière suivante :*

$$LOAD(k) = \max_{t>0} \left(\frac{\sum_{j=1}^{k} DBF(\tau_j, t)}{t} \right)$$

Considérons τ un ensemble de tâches sporadiques dont les priorités sont assignées par un algorithme à priorités fixes. Considérons un scénario où une instance de la tâche τ_k commet un dépassement d'échéance sur le processeur π_j. Nous notons t_a la date à laquelle cette échéance est activée et t_d la date à laquelle se produit le dépassement.

$$t_d - t_a = D_k \tag{5.5}$$

Considérons $W(t_a)$ la durée d'exécution des instances qui ont une échéance $\leq t_d$ et qui sont exécutées sur l'intervalle $[t_a, t_d)$. *r-SP_wl* a assigné l'instance de τ_k qui dépasse son échéance à l'instant t_a sur le processeur π_j car celui-ci avait la plus grande valeur de laxité minimale. Mais une instance plus prioritaire a été activée avant t_d, ce qui conduit au dépassement d'échéance de l'instance de τ_k. Ce processeur où a été assignée l'instance de τ_k a donc exécuté des instances pour une durée $t_d - t_a$ (il n'a pas été oisif). La seule hypothèse que nous pouvons faire sur les autres processeurs est qu'au moins une instance est active à tout instant. En effet, si un processeur est inoccupé, sa laxité sera considérée comme infinie, et il sera forcément choisi pour l'activation d'une future instance. Nous pouvons faire l'hypothèse la plus pessimiste que les $(m-1)$ autres processeurs exécutent les instances qui ont la plus petite densité. Ce qui nous conduit à :

$$W(t_a) > (t_d - t_a) + (m-1) \times (t_d - t_a) \times \delta_{min}(k) \tag{5.6}$$

Parce que $W(t_a)$ correspond à l'exécution d'instances sur l'intervalle $[t_a, t_d)$, toutes les instances contribuant à $W(t_a)$ doivent avoir une fenêtre d'exécution dont l'intersection avec $[t_a, t_d)$ est non nulle. Pour cela, nous considérons les instances activées après $t_a - D_{max}(k)$ dont l'échéance précède $t_d + D_{max}(k)$. Toutes ces instances ont leur activation et leur échéance dans un intervalle de taille $(2D_{max}(k) + (t_d - t_a))$. Par définition du paramètre *LOAD*, nous avons :

$$\begin{aligned}
&W(t_a) \leq (t_d - t_a + 2D_{max}(k)) \times LOAD(k) \\
&\Rightarrow (\textit{inégalité } 5.6)\, (t_d - t_a) + (m-1)(t_d - t_a)\delta_{min}(k) < (t_d - t_a + 2D_{max}(k)) \times LOAD(k) \\
&\equiv (t_d - t_a)(1 + (m-1)\delta_{min}(k)) < (t_d - t_a + 2D_{max}(k)) \times LOAD(k) \\
&\equiv 1 + (m-1)\delta_{min}(k) < \left(1 + 2\frac{D_{max}(k)}{t_d - t_a}\right) \times LOAD(k) \\
&\Rightarrow (\textit{égalité } 5.5)\, 1 + (m-1)\delta_{min}(k) < \left(1 + 2\frac{D_{max}(k)}{D_k}\right) \times LOAD(k)
\end{aligned}$$

Cette condition est une condition nécessaire pour qu'un dépassement d'échéance soit commis. La négation de cette condition nous donne une condition suffisante de faisabilité.

Proposition 5.7. *Une condition suffisante pour qu'un ensemble de tâche soit ordonnançable avec l'algorithme r-SP_wl est donnée par :*

$$\forall k : 1 \leq k \leq n : \left[LOAD(k) \leq \frac{1 + (m-1)\delta_{min}(k)}{1 + 2(D_{max}(k)/D_k)} \right]$$

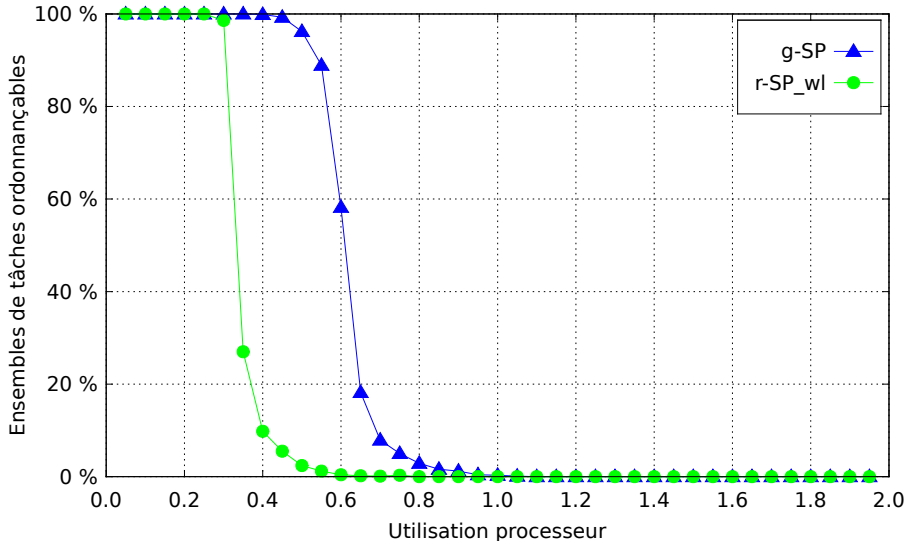


FIGURE 5.6 – Comparaison des bornes basées sur le LOAD de l'algorithme global et de r-SP_wl.

Dans la figure 5.6, nous comparons le test d'ordonnançabilité basé sur le LOAD de l'algorithme global avec celui de l'algorithme *r-SP_wl*. Ces tests ont été implantés en utilisant un algorithme d'approximation pour calculer le LOAD comme décrit par Fisher, Baker et Baruah dans [FBB06a]. Les paramètres des simulations sont les mêmes que ceux utilisés pour la simulation de la figure 5.5. Nous remarquons que l'écart entre les courbes représentant l'ordonnançabilité de l'algorithme global et de l'algorithme *r-SP_wl* est bien plus important pour la figure 5.6 que pour la figure 5.5. En effet, il est plus difficile de construite un scénario où une instance dépasse son échéance avec *r-SP_wl*.

5.6 Conclusion

Dans ce chapitre, nous avons décrit notre algorithme *r-SP_wl*. Il est conçu pour garantir la prédictibilité de l'ordonnancement. Nous avons montré qu'il était viable dans le cas des tâches périodiques, avec la perspective d'explorer le cas des tâches sporadiques. La prédictibilité nous permet de proposer un intervalle d'étude qui nous donne

un test nécessaire et suffisante d'ordonnançabilité. Cet intervalle pouvant être très grand et donc l'ordonnancement sur cet intervalle pouvant être très long à construire, nous proposons un test d'ordonnançabilité suffisant. Une perspective est d'explorer d'autres techniques de construction de conditions suffisantes d'ordonnançabilité dans le but d'obtenir une meilleure borne. Ce travail a donné lieu aux publications [FGMM11, FMG11]

CHAPITRE 6

SIMULATEUR

6.1 Introduction

La simulation est un approche nécessaire pour comparer plusieurs algorithmes d'ordonnancement ou plusieurs tests d'ordonnançabilité. Elle permet de générer aléatoirement des ensembles de tâches et d'évaluer les propriétés des algorithmes. Bien que les systèmes temps réel soient étudiés depuis longtemps, aucun outil de simulation libre n'a réussi à s'imposer comme standard dans le monde académique (comme peut l'être *ns-2* pour la simulation des protocoles réseaux). Cheddar [SJNM04], bien qu'étant un outil de simulation abouti, nécessite la connaissance de la programmation en langage Ada. Même si nous n'avons pas la prétention de fournir un tel standard, nous avons décidé de rendre disponible le code développé dans le cadre de cette thèse sous la forme du logiciel libre Real-Time Multiprocessor SIMulator (RTMSim).

6.2 Propriétés du logiciel

6.2.1 Licence

RTMSim est développé et distribué sous *licence BSD modifiée*. Il s'agit d'une licence libre qui permet à quiconque de pouvoir exploiter le code de ce logiciel, de le modifier et de le redistribuer sous une quelconque forme. Cette licence n'est pas contaminante car elle n'impose pas au contributeur de continuer à utiliser la même licence. Ce choix a été motivé par la volonté de ne mettre aucune entrave juridique pour pouvoir utiliser ou contribuer au développement de RTMSim.

6.2.2 Langage de programmation

RTMSim est développé avec le langage de programmation *Java*. *Java* est un langage objet de haut niveau et utilisé par une large communauté de développeurs. Nous expliquons par la suite quelles propriétés font de RTMSim un simulateur évolutif.

6.2.3 Interface utilisateur

Le développement d'une interface graphique est une tâche longue et qui apporte que de valeur ajoutée à un logiciel de simulation. Si l'interface graphique permet de rendre compte rapidement du potentiel d'un logiciel, elle est souvent difficilement maintenable pour un logiciel en cours de développement. Nous nous sommes focalisés sur le développement d'un noyau de simulateur évolutif. C'est pourquoi nous avons fait le choix d'utiliser une interface en ligne de commande. Plutôt que de fournir une commande monolithique difficile à prendre en main, nous avons préféré respecter la philosophie des systèmes *Unix* consistant à ce qu'une commande ne fournisse qu'une seule fonctionnalité. La commande doit être simple d'utilisation et pouvoir interagir avec le reste du système. RTMSim est donc constitué d'un ensemble de commandes qui peuvent être liées entre elles. Nous donnons l'exemple de la visualisation d'un diagramme d'ordonnancement. RTMSim dispose des commandes `taskgen`, permettant d'engendrer aléatoirement des ensembles de tâches, `sched`, permettant de simuler un ordonnancement et `schedviz`, permettant de visualiser la trace d'un ordonnancement. Le diagramme de l'ordonnancement global d'un ensemble de tâches engendré aléatoirement pourra être obtenu en exécutant :

```
java -jar taskgen.jar | java -jar sched.jar -a GLOBAL_DM | java -jar
schedviz.jar
```

La syntaxe « | » permet d'établir un tube de communication qui relie la sortie standard d'une commande à l'entrée standard de la commande suivante.

6.2.4 Format des données

Les données produites (traces, résultats) ou traitées (jeux de tâches) par les commandes de RTMSim utilisent le format *eXtensible Markup Language (XML)*. Il s'agit d'un langage de balises permettant de mettre en forme toute sorte de données. Il existe plusieurs analyseurs syntaxiques pour XML, évitant ainsi de devoir écrire un analyseur spécifique pour un format de données non standard. Ce choix a été motivé par la volonté de rendre notre format de données facilement accessible en utilisant un langage largement exploité et reconnu.

6.2.5 Validation

La validation de notre simulateur s'effectue grâce à un système de tests unitaires. Ces tests permettent de vérifier à partir de scénarios connus (*use cases*) que les algorithmes implantés produisent bien le résultat attendu. Cette technique permet de tester indépendamment les différentes briques logicielles qui forment RTMSim et de se prémunir contre des régressions pouvant survenir durant le développement.

6.3 Architecture

Le langage *Java* nous a permis de fournir à RTMSim une architecture évolutive, notamment grâce au concept d'*héritage*, d'*interface* et de *polymorphisme*. L'*héritage* permet à un objet « père » de transmettre ses caractéristiques à un objet « fils » qui pourra ainsi les étendre. Une tâche périodique est caractérisée, entre autre, par son WCET, son échéance et sa période. Une tâche périodique dépendante partage les mêmes caractéristiques qu'une tâche périodique avec la particularité qu'un ensemble de sections critiques lui est associé. Ainsi l'objet `DependentTask` *hérite* de l'objet `Task`. Une *interface* permet de spécifier un comportement commun à différents objets. Nous considérons un objet ordonnançable si il est caractérisé par un instant d'activation, une durée d'exécution et une échéance. Ainsi, un objet `Task` est un objet de type `Schedulable` et implante les méthodes `getStart()`, `getWCET()` et `getDeadline()`. Le *polymorphisme* permet, grâce à l'*héritage*, que des objets différents aient le même comportement. Ainsi, les objets `Task` et `DependentTask` qui sont tous deux de type `Schedulable`, implantent la méthode `getWCET()`. Mais tandis que l'objet `Task` ne fait que retourner la valeur d'un champs `wcet` qui caractérise sa durée d'exécution, l'objet `DependentTask` retourne la somme des durées d'exécution de ses segments (segments synchronisés et segments non synchronisés).

6.3.1 Génération de tâches

Comparer par simulation des algorithmes sur des données totalement aléatoires revient à comparer leur comportement en moyenne. Pour limiter l'introduction de biais dans la simulation, il est nécessaire d'engendrer une quantité importante de données. Dans le cas des systèmes temps réel, Bini et Buttazzo ont proposé l'algorithme *UUniFast* qui permet d'engendrer aléatoirement et sans biais un ensemble de tâches [BB04]. Initialement proposé dans le cas ou l'utilisation de l'ensemble de tâches est bornée par 1 (cas monoprocesseur), il a été adapté (algorithme *UUniFast-Discard*) par Davis et Burns [DB10] pour le cas des systèmes multiprocesseurs. Cet algorithme permet d'engendrer un ensemble de n tâches pour lesquelles l'utilisation est uniformément répartie en rejetant les ensembles contenant des tâches ayant une utilisation supérieure à 1. Pour un nombre de processeurs important, cette technique conduit à un

grand nombre de rejet. Bertogna a aussi proposé un algorithme pour engendrer des tâches pour les systèmes multiprocesseurs [Ber08] qui ne produit pas de rejet comme *UUniFast-Discard*. Mais cet algorithme combine deux variables (l'utilisation et la cardinalité des ensembles de tâches) et n'engendre donc pas nécessairement des systèmes exempts de biais.

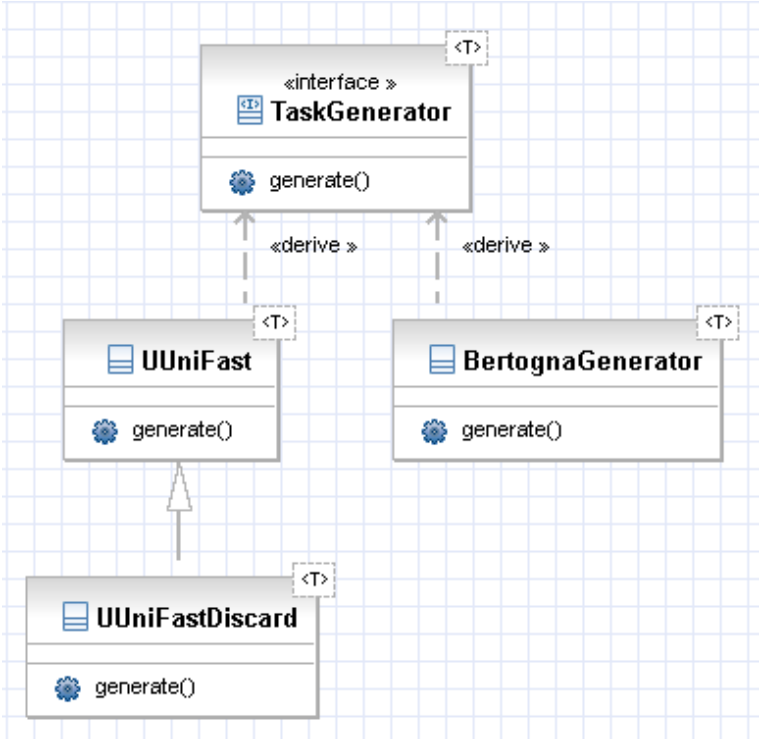


FIGURE 6.1 – Diagramme de représentation des algorithmes de génération de tâches.

Ces deux algorithmes sont complémentaires. L'un permet de garantir une absence de biais mais est limité par le nombre de processeurs tandis que l'autre peut produire des ensembles de tâches pour un nombre important de processeurs mais ne garantit pas l'absence de biais. Nous les avons donc implantés avec une interface commune comme représenté dans la figure 6.1. Il devient alors possible de choisir quel algorithme sera utilisé en exécutant la commande `taskgen` avec en paramètre le nom de la classe.

6.3.2 Algorithmes pour la robustesse

Nous avons présenté dans le chapitre 2 deux approches algorithmiques pour calculer la marge sur le WCET ou la fréquence. Le calcul d'*Allowance* proposé par Bougueroua, George et Midonnet [BGM07] consiste à trouver par recherche dichotomique la plus grande valeur de marge. L'analyse de sensibilité proposé par Bini, Di Natale et Buttazzo [BDNB06] recherche cette valeur parmi un ensemble d'instants d'ordonnancement. Même si leur formalisme diffère, ces deux algorithmes permettent de calculer les mêmes valeurs de marge maximale. Le calcul d'*Allowance* explorera dans la plupart des cas un ensemble de valeur plus grand que l'analyse de sensibilité. Mais le

calcul des instants d'ordonnancement, dans le cas de l'analyse de sensibilité, a une complexité exponentielle en le nombre de tâches, ce qui le rend difficilement exploitable pour de grands ensembles de tâches.

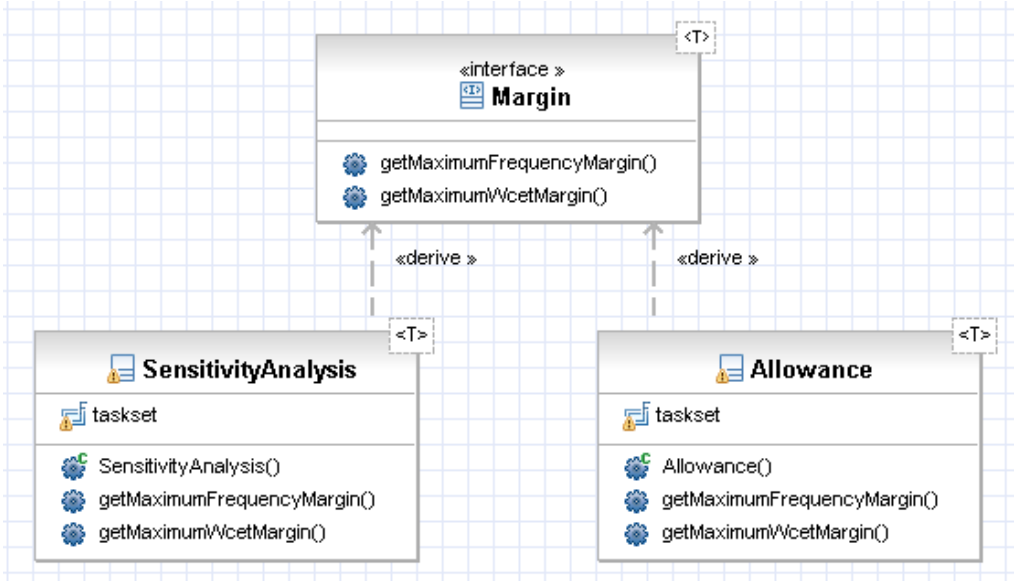


FIGURE 6.2 – Diagramme de représentation des algorithmes de calcul de marge.

Nous les avons donc implantés avec une interface commune comme représenté dans la figure 6.2. Le choix de l'algorithme de calcul de la marge revient à l'utilisateur.

6.3.3 Algorithmes de partitionnement

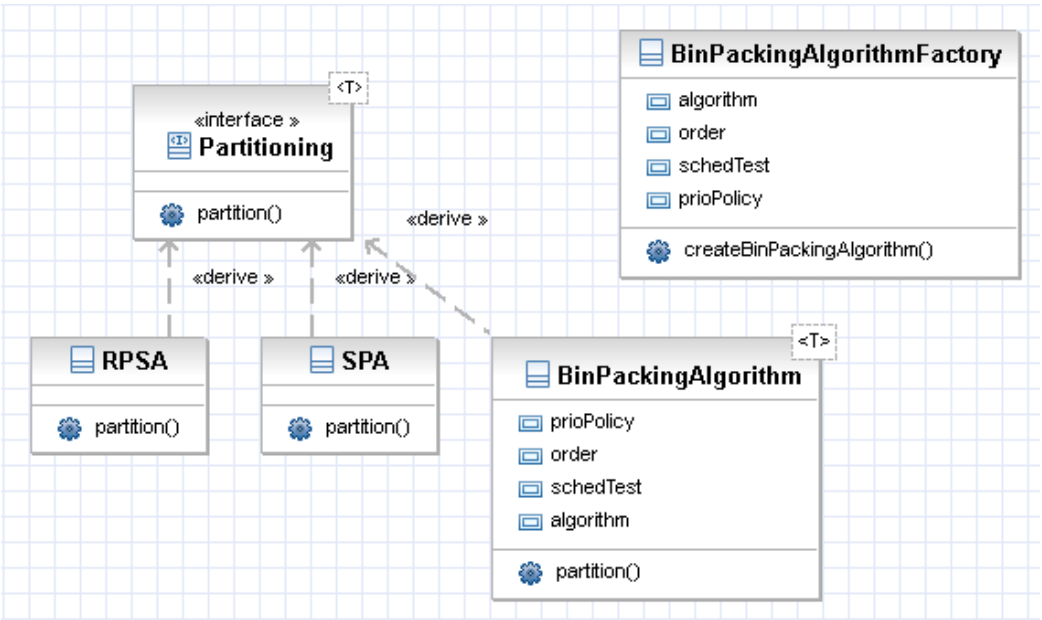


FIGURE 6.3 – Diagramme de représentation des algorithmes de partitionnement.

Comme nous l'avons présenté dans le chapitre 3, le partitionnement d'un ensemble de tâches consiste à subdiviser cet ensemble en autant de sous-ensembles que de processeurs. Il est donc naturel de définir une *interface* pour définir ce comportement. Plutôt que d'implanter tous les algorithmes de partitionnement basés sur les algorithmes pour BIN-PACKING, nous avons défini un objet abstrait `BinPackingAlgorithm`. Il représente le comportement d'un algorithme pour BIN-PACKING. Il utilise des objets dédiés pour (i) le tri des tâches (DU, DD, IP, ...), (ii) l'algorithme de choix du processeur (FF, BF, NF, ...), (iii) la politique d'attribution des priorités et (iv) le test d'ordonnançabilité. L'objet `BinPackingAlgorithmFactory` est un constructeur permettant de produire un algorithme de partitionnement de deux façons différentes. Soit il utilise une description du comportement de l'algorithme à partir des objets dédiés (i), (ii), (iii) et (iv), soit il utilise la dénomination d'un algorithme connu (`FBB_FFD` est par exemple décrit par (i) ID, (ii) FF, (iii) DM et (iv) un test basé sur la *Request Bound Function (RBF)*). Les algorithmes RPSA (chapitre 4) et SPA [LdNR09] sont des algorithmes de partitionnement pour tâches dépendantes qui ont un comportement différent des algorithmes de BIN-PACKING. Ils disposent de leur propre classe. Tous ces algorithmes sont donc implantés suivant la hiérarchie de classe décrite dans la figure 6.3.

6.3.4 Algorithmes d'ordonnancement

Nous avons implanté un simulateur d'ordonnancement pour évaluer les performances en terme d'ordonnançabilité de l'algorithme proposé dans le chapitre 5. Il est initialisé par les activations des tâches qui ajoute des évènements dans une file événementielle. Un évènement correspond à un changement dans le système et peut représenter une activation d'instance, une terminaison d'instance, une préemption, ou tout autre action susceptible d'influer sur l'ordonnancement. Ces évènements sont extraits de la pile puis traités, et leur traitement génère de nouveaux évènements jusqu'à ce que la fin de la simulation soit atteinte. Le traitement des évènements permet le déroulement de l'ordonnancement, mais aussi la création d'une trace qui peut être exploitée, notamment pour la création d'un diagramme d'ordonnancement. Durant le développement, nous avons exploité les similitudes entre les différentes approches d'ordonnancement. Plutôt que d'écrire une classe pour chaque algorithme, nous avons préféré développer une unique classe `Scheduler` qui représente le comportement d'un ordonnanceur événementiel. Cet objet est paramétré par une politique de gestion des priorités, ainsi que par une politique de migrations. L'ordonnanceur délègue alors le traitement des évènements à cette dernière.

Nous avons implanté trois ordonnanceurs en respectant le modèle de conception décrit dans la figure 6.4 :

- l'algorithme (FTP-FJII) décrit par Ha et Liu [HL94] ;
- *r-SP_wl* ;

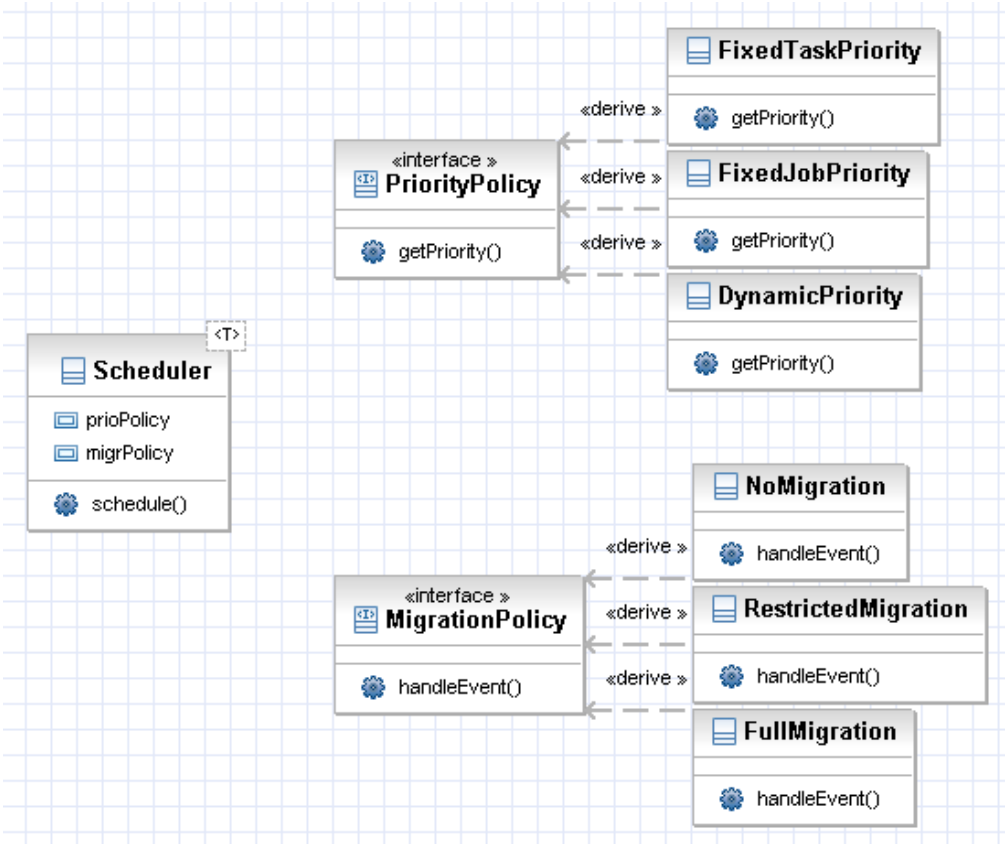


FIGURE 6.4 – Diagramme de représentation des algorithmes d'ordonnancement.

– l'algorithme (FTP-DII) (global priorités fixes).

Ils appliquent une politique de gestion de priorités héritant de `FixedTaskPriority`. L'algorithme à migrations restreintes de Ha et Liu et l'algorithme *r-SP_wl* appliquent une politique de migrations restreintes , héritant de la classe `RestrictedMigration`. L'algorithme (FTP-DII) applique une politique de migrations non restreintes en héritant de la classe `FullMigration`.

6.4 Conclusion

RTMSim n'a pas la prétention d'être un produit fini. Il s'agit d'un outil qui a été développé dans un cadre académique avec pour objectif de fournir rapidement des intuitions sur la pertinence de différents algorithmes d'ordonnancement. Mais plutôt que d'investir du temps à produire des programmes ne pouvant être réutilisés par la suite (car non évolutifs), nous avons fait un effort de développement pour fournir un ensemble cohérent qui, s'il ne devient pas un logiciel de référence, offrira une base solide pour un effort coopératif plus vaste. Une perspective est de s'inspirer du modèle de conception de RTMSim pour proposer des extensions à la spécification temps réel pour Java (*Real-time Specification for Java*) (RTSJ) pour les systèmes multiprocesseurs.

CHAPITRE 7

CONCLUSION GÉNÉRALE

L'étude de l'ordonnancement monoprocesseur de tâches temps réel a conduit à plusieurs approches algorithmiques pour décider de l'ordre d'exécution des tâches. Il est souvent fait référence à l'ordonnancement à priorités fixes et à l'ordonnancement à priorités dynamiques (EDF, LLF), même si nous préférons une terminologie plus rigoureuse de classe d'ordonnancement FTP, FJP et DP qui fait intervenir la notion d'instances de tâche. L'avènement des plate-formes multi-cœur grand public a relancé l'intérêt pour la problématique de l'ordonnancement temps réel multiprocesseur. L'étude de l'ordonnancement monoprocesseur semble avoir atteint une certaine maturité, notamment avec des ordonnanceurs optimaux tels qu'EDF qui sont désormais implantés sur des systèmes d'exploitation comme Linux [FTC09]. Même si Linux n'est pas un système d'exploitation temps réel strict, plusieurs projets, notamment RTAI et Xenomai, apportent des solutions pour l'ordonnancement temps réel strict à priorités fixes. Pourtant, l'étude de l'ordonnancement multiprocesseur a encore du chemin à parcourir. En effet, le degré de liberté supplémentaire qu'apporte la migration interprocesseur augmente considérablement le nombre d'approches d'ordonnancement. Il est alors possible d'avoir des ordonnanceurs par partitionnement, globaux ou encore hybrides entre ces deux approches. Même si des algorithmes d'ordonnancement optimaux sont identifiés depuis une quinzaine d'années, ils reposent souvent sur des modèles qui rendent leur implantation difficile.

Dans ce manuscrit, nous avons abordé cette problématique de l'ordonnancement temps réel multiprocesseur du point de vue de la sûreté temporelle. Nous définissons la sûreté temporelle comme d'une part la robustesse temporelle consistant à tolérer des dépassements de WCET et des augmentations de fréquence d'activation des tâches, et d'autre part, la viabilité. La viabilité consiste à garantir qu'un algorithme ou qu'un test d'ordonnançabilité soit résistant à la diminution des WCET, l'augmentation des échéances et l'augmentation des périodes qui peuvent parfois conduire à des anomalies d'ordonnancement. Un exemple d'anomalie serait un algorithme capable d'ordon-

nancer un ensemble de tâches sans qu'aucune ne dépasse d'échéance, mais qui ne pourrait pas ordonnancer l'ensemble de tâches pour lequel une instance aurait une échéance plus grande.

Nous avons concentré notre attention sur la classe d'ordonnancement FTP. Nous avons étudié l'approche par partitionnement (classe FTII) pour des tâches sporadiques et indépendantes. Cette étude a consisté en une comparaison de plusieurs d'algorithmes de partitionnement. Nous avons identifié ceux qui permettent de maximiser la marge sur le WCET et sur la fréquence. Nous avons ensuite élargi cette étude au cas des tâches dépendantes qui partagent des ressources. Ce modèle de tâche implique l'utilisation d'un protocole de synchronisation pour garantir un temps de blocage borné. Il devient alors plus difficile de construire un algorithme car la maximisation de la marge n'a plus de lien direct avec la minimisation des temps de blocage, alors qu'elle en avait un avec l'ordonnançabilité.

Toujours en contexte d'ordonnancement FTP, nous nous sommes intéressé aux ordonnancements avec migrations. Nous avons proposé un algorithme d'ordonnancement à migrations restreintes, pour lequel une tâche n'est autorisée à migrer que lors de l'activation de ses instances. Il a été prouvé qu'un algorithme appartenant à cette classe d'ordonnancement était sujet à des anomalies relatives aux diminutions de WCET. Nous avons donc proposé un algorithme d'ordonnancement viable pour les tâches périodiques en permettant que cet ordonnancement soit oisif. Nous avons également proposé un intervalle d'étude et condition suffisante d'ordonnançabilité pour cet algorithme.

Nous avons développé un simulateur nous permettant d'évaluer toutes les approches que nous avons proposées. Les sources de ce logiciel sont librement disponibles.

Une perspective à court terme est l'implantation de ces travaux dans un système d'exploitation temps réel. L'implantation ne doit bien sûr pas être une fin en soit mais bien un exercice technique qui permet de mettre en évidence les imperfections des modèles théoriques ou la trop grande rigidité des hypothèses de départ.

Si l'ordonnancement parallèle d'une tâche est depuis longtemps étudié pour les systèmes sans contraintes temps réel, l'intérêt pour le parallélisme dans les systèmes temps réel va grandissant. En effet, de récents travaux commencent à s'intéresser à cette problématique et étendent des résultats connus de l'ordonnancement temps réel multiprocesseur vers l'ordonnancement parallèle avec contraintes de temps [CCG08, KI09, GB10, LKRR10, FMQ11, QFM11]. Une autre perspective, qui pourra faire l'objet d'un travail théorique, est l'étude de la sûreté temporelle dans les systèmes de tâches parallèles.

BIBLIOGRAPHIE

[AATW93] Neil C. AUDSLEY, Burns ALAN, Ken W. TINDELL et Andy J. WELLINGS :
Applying new scheduling theory to static priority pre-emptive scheduling.
Software Engineering Journal, 8(5):284–292, Sepember 1993.

[ABB08] Björn ANDERSSON, Konstantinos BLETSAS et Sanjoy K. BARUAH :
Scheduling arbitrary-deadline sporadic task systems on multiprocessors.
In Proceedings of the 29th IEEE Real-Time Systems Symposium (RTSS), pages 385–394, Barcelona, Spain, November - December 2008. IEEE Computer Society.

[ABD08] James H. ANDERSON, Vasile BUD et UmaMaheswari C. DEVI :
An EDF-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems.
Real-Time Systems, 38(2):85–131, 2008.

[ABJ01] Björn ANDERSSON, Sanjoy K. BARUAH et Jan JONSSON :
Static-priority scheduling on multiprocessors.
In Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS), pages 193–202, London, UK, December 2001. IEEE Computer Society.

[AJ02] Björn ANDERSSON et Jan JONSSON :
Preemptive multiprocessor scheduling anomalies.
In Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS), pages 12–19, Fort Lauderdale, Florida, USA, April 2002. IEEE Computer Society.

[AS00] James H. ANDERSON et Anand SRINIVASAN :
Early-release fair scheduling.
In Proceedings of the 12th Euromicro Conference on Real-time Systems (ECRTS), page 35–43, Stockholm, Sweden, June 2000. IEEE Computer Society.

[AT06] Björn ANDERSSON et Eduardo TOVAR :

Multiprocessor scheduling with few preemptions.
In Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), page 322–334, Sydney, Australia, August 2006. IEEE Computer Society.

[Aud01] Neil C. AUDSLEY :
On priority assignment in fixed priority scheduling.
Information Processing Letters, 79:39–44, 2001.

[BA08] Björn B. BRANDENBURG et James H. ANDERSON :
An implementation of the PCP, SRP, D-PCP, M-PCP, and FMLP real-time synchronization protocols in $LITMUS^{RT}$.
In Proceedings of the 14th IEEE International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA), pages 185–194, Kaohsiung, Taiwan, August 2008. IEEE Computer Society.

[Bak91] Theodore P. BAKER :
Stack-based scheduling of realtime processes.
Real-Time Systems, 3(1):67–99, March 1991.

[Bak03] Theodore P. BAKER :
Multiprocessor EDF and deadline monotonic schedulability analysis.
In Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS), pages 120–129, Tallahassee, FL, USA, December 2003. IEEE Computer Society.

[Bak05] Theodore P. BAKER :
Comparison of empirical success rates of global vs. partitioned fixed-priority and EDF scheduling for hard real time.
Rapport technique TR-050601, Florida State University, Tallahassee, FL, USA, July 2005.

[Bar07] Sanjoy K. BARUAH :
Techniques for multiprocessor global schedulability analysis.
In Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS), pages 119–128, Tucson, Arizona, USA, December 2007. IEEE Computer Society.

[BB04] Enrico BINI et Giorgio C. BUTTAZZO :
Biasing effects in schedulability measures.
In Proceedings of the 16th Euromicro Conference on Real-time Systems (ECRTS), pages 196–203, Catania, Sicily, Italy, June - July 2004. IEEE Computer Society.

[BB06] Sanjoy K. BARUAH et Alan BURNS :
Sustainable scheduling analysis.

In Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), page 159–168, Rio de Janeiro, Brazil, December 2006. IEEE Computer Society.

[BB08] Alan BURNS et Sanjoy K. BARUAH :
Sustainability in real-time scheduling.
Journal of Computing Science and Engineering (JCSE), 2(1):74–97, March 2008.

[BB09a] Theodore P. BAKER et Sanjoy K. BARUAH :
An analysis of global EDF schedulability for arbitrary-deadline sporadic task systems.
Real-Time Systems, 43(1):3–24, September 2009.

[BB09b] Theodore P. BAKER et Sanjoy K. BARUAH :
Sustainable multiprocessor scheduling of sporadic task systems.
In Proceedings of the 21st Euromicro Conference on Real-time Systems (ECRTS), pages 141–150, Dublin, Irland, July 2009. IEEE Computer Society.

[BBMSS10] Sanjoy K. BARUAH, Vincenzo BONIFACI, Alberto MARCHETTI-SPACCAMELA et Sebastian STILLER :
Improved multiprocessor global schedulability analysis.
Real-Time Systems, 46(1):3–24, September 2010.

[BC03] Sanjoy K. BARUAH et John CARPENTER :
Multiprocessor fixed-priority scheduling with restricted interprocessor migrations.
In Proceedings of the 15th Euromicro Conference on Real-time Systems (ECRTS), pages 195–202, Porto, Portugal, July 2003. IEEE Computer Society.

[BC07] Marko BERTOGNA et Michele CIRINEI :
Response-time analysis for globally scheduled symmetric multiprocessor platforms.
In Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS), pages 149–160, Tucson, Arizona, USA, December 2007. IEEE Computer Society.

[BCL09] Marko BERTOGNA, Michele CIRINEI et Giuseppe LIPARI :
Schedulability analysis of global scheduling algorithms on multiprocessor platforms.
IEEE Transactions on Parallel and Distributed Systems, 20(4):553–566, April 2009.

[BCPV96] Sanjoy K. BARUAH, N. K. COHEN, Charles Gregory PLAXTON et D. A. VARVEL :
Proportionate progress : A notion of fairness in resource allocation.

Algorithmica, 15(6):600–625, June 1996.

[BDNB06] Enrico BINI, Marco DI NATALE et Giorgio C. BUTTAZZO :
Sensitivity analysis for fixed-priority real-time systems.
In Proceedings of the 18th Euromicro Conference on Real-time Systems (ECRTS), pages 13–22, Dresden, Germany, April 2006. IEEE Computer Society.

[Ber08] Marko BERTOGNA :
Real-Time Scheduling Analysis for Multiprocessor Platforms.
Thèse de doctorat, Scuola Superiore Sant'Anna, Pisa, Italy, May 2008.

[BF08] Sanjoy K. BARUAH et Nathan Wayne FISHER :
Global fixed-priority scheduling of arbitrary-deadline sporadic task systems.
In Shrisha RAO, Mainak CHATTERJEE, Prasad JAYANTI, C. Siva Ram MURTHY et Sanjoy Kumar SAHA, éditeurs : *Proceedings of the 9th International Conference on Distributed Computing and Networking (ICDCN)*, volume 4904/2008, pages 215–226, Kolkota, India, January 2008. Springer.

[BGM07] Lamine BOUGUEROUA, Laurent GEORGE et Serge MIDONNET :
Dealing with execution-overruns to improve the temporal robustness of real-time systems scheduled FP and EDF.
In Proceedings of the 2nd International Conference on Systems (ICONS), page 8pp, Sainte-Luce, Martinique, April 2007. IEEE Computer Society.

[BLBA07] Aaron David BLOCK, Hennadiy LEONTYEV, Björn B. BRANDENBURG et James H. ANDERSON :
A flexible real-time locking protocol for multiprocessors.
In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 47–56, Daegu, South Korea, August 2007. IEEE Computer Society.

[BLOS95] Almut BURCHARD, Jörg LIEBEHERR, Yingfeng OH et Sang H. SON :
New strategies for assigning real-time tasks to multiprocessor systems.
IEEE Transactions on Computers, 44(12):1429–1442, December 1995.

[BMR90] Sanjoy K. BARUAH, Aloysius Ka-Lau MOK et Louis E. ROSIER :
Preemptively scheduling hard-real-time sporadic tasks on one processor.
In Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS), pages 182–190, Orlando, Florida, USA, December 1990. IEEE Computer Society.

[CCG08] Sébastien COLETTE, Liliana CUCU et Joël GOOSSENS :
Integrating job parallelism in real-time scheduling theory.
Information Processing Letters, 106(5):180–187, May 2008.

[CFH+04] John CARPENTER, Shelby Hyatt FUNK, P. HOLMAN, A. SRINIVASAN, James H. ANDERSON et Sanjoy K. BARUAH :
Handbook of Scheduling : Algorithms, Models, and Performance Analysis, chapitre A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms, pages 30–1 – 30–19.
Chapman and Hall/CRC, Boca Raton, Florida, 2004.

[CG06] Liliana CUCU et Joël GOOSSENS :
Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors.
In Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation, pages 397–405, Prague, Czech Republic, September 2006.

[CP01] Antoine COLIN et Isabelle PUAULT :
Worst-case execution time analysis of the RTEMS real-time operating system.
In Proceedings of the 13th Euromicro Conference on Real-time Systems (ECRTS), pages 191–198, Delft , Netherlands, June 2001. IEEE Computer Society.

[CRJ06] Hyeonjoong CHO, Binoy RAVINDRAN et E. Douglas JENSEN :
An optimal real-time scheduling algorithm for multiprocessors.
In Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS), pages 101–110, Rio de Janeiro, Brazil, December 2006. IEEE Computer Society.

[DB95] Robert Ian DAVIS et Alan BURNS :
Optimal priority assignment for aperiodic tasks with firm deadlines in fixed priority pre-emptive systems.
Information Processing Letters, 53(5):249 – 254, March 1995.

[DB07] Robert Ian DAVIS et Alan BURNS :
Robust priority assignment for fixed priority real-time systems.
In Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS), pages 3–14, Tucson, Arizona, USA, December 2007. IEEE Computer Society.

[DB10] Robert Ian DAVIS et Alan BURNS :
Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems.
Real-Time Systems, 47(1):1–40, 2010.

[DB11] Robert Ian DAVIS et Alan BURNS :
FPZL schedulability analysis.
In Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 245–256, Chicago, IL, USA, April 2011. IEEE Computer Society.

[DD86] Sadegh DAVARI et Sudarshan K. DHALL :
An on line algorithm for real-time tasks allocation.
In Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS), pages 194–200, New Orleans, LA,USA, December 1986. IEEE Computer Society.

[Der74] Michael Leonidas DERTOUZOS :
Control robotics : The procedural control of physical processes.
In Jack L. ROSENFELD, éditeur : *Proceedings of IFIP Congress 74*, pages 807–813, Stockholm, Sweden, August 1974. North-Holland, American Elsevier.

[DL78] Sudarshan K. DHALL et Chung Laung LIU :
On a real-time scheduling problem.
Operations Research, 26(1):127–140, January-February 1978.

[DM89] Michael Leonidas DERTOUZOS et Aloysius Ka-Lau MOK :
Multiprocessor on-line scheduling of hard-real-time tasks.
IEEE Transactions on Software Engineering, 15(12):1497–1506, December 1989.

[DMYGR10] François DORIN, Patrick MEUMEU YOMSI, Joël GOOSSENS et Pascal RICHARD :
Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms.
June 2010.

[DNS95] Marco DI NATALE et John A. STANKOVIC :
Applicability of simulated annealing methods to real-time scheduling and jitter control.
In Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS), pages 190–199, Pisa, Italy, December 1995. IEEE Computer Society.

[FB05] Shelby Hyatt FUNK et Sanjoy K. BARUAH :
Restricting EDF migration on uniform heterogeneous multiprocessors.
TSI. Technique et science informatiques, 24(8):917–938, 2005.

[FBB06a] Nathan Wayne FISHER, Theodore P. BAKER et Sanjoy K. BARUAH :
Algorithms for determining the demand-based load of a sporadic task system.
In Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 135–146, Sydney, Australia, August 2006. IEEE Computer Society.

[FBB06b] Nathan Wayne FISHER, Sanjoy K. BARUAH et Theodore P. BAKER :
The partitioned scheduling of sporadic tasks according to static-priorities.

In Proceedings of the 18th Euromicro Conference on Real-time Systems (ECRTS), pages 118–127, Dresden, Germany, July 2006. IEEE Computer Society.

[FGMM11] Frédéric FAUBERTEAU, Laurent GEORGE, Damien MASSON et Serge MIDONNET :
Ordonnancement multiprocesseur global basé sur la laxité avec migrations restreintes.
In Proceedings of the 12th Congrès Annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF), pages 47–48, Saint-Étienne, France, March 2011.

[Fis07] Nathan Wayne FISHER :
The Multiprocessor Real-Time Scheduling of General Task Systems.
Thèse de doctorat, University of North Carolina at Chapel Hill, 2007.

[FMG09] Frédéric FAUBERTEAU, Serge MIDONNET et Laurent GEORGE :
Allowance-fit : A partitioning algorithm for temporal robustness of hard real-time systems upon multiprocessors.
In Proceedings of the Work-in-Progress Session of the 14th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), page 4pp, Mallorca, Spain, September 2009. IEEE Computer Society.

[FMG10a] Frédéric FAUBERTEAU, Serge MIDONNET et Laurent GEORGE :
A robust partitioned scheduling for real-time multiprocessor systems.
In Proceedings of the 7th IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES), pages 193–204, Brisbane, Australia, September 2010. Springer Science and Business Media.

[FMG10b] Frédéric FAUBERTEAU, Serge MIDONNET et Laurent GEORGE :
Robust partitioned scheduling for static-priority real-time multiprocessor systems with shared resources.
In Proceedings of the 18th International Conference on Real-Time and Network Systems (RTNS), pages 217–225, Toulouse, France, November 2010.

[FMG11] Frédéric FAUBERTEAU, Serge MIDONNET et Laurent GEORGE :
Laxity-based restricted-migration scheduling.
In Proceedings of the 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), page 8pp., Toulouse, September 2011.

[FMQ11] Frédéric FAUBERTEAU, Serge MIDONNET et Manar QAMHIEH :
Partitioned scheduling of parallel real-time tasks on multiprocessor systems.
ACM SIGBED Review, 8(3):4pp, September 2011.
Special Issue on Work-in-Progress (WiP) session of the 23rd Euromicro Conference on Real-Time Systems (ECRTS 2011).

[FTC09] Dario FAGGIOLI, Michael TRIMARCHI et Fabio CHECCONI :
An implementation of the earliest deadline first algorithm in linux.
In Proceedings of 24th ACM Symposium on Applied Computing (SAC), pages 1984–1989, Manoa, HI, USA, March 2009.

[GB10] Joël GOOSSENS et Vandy BERTEN :
Gang FTP scheduling of periodic and parallel rigid real-time tasks.
In Proceedings of the 18th Real-Time and Network Systems (RTNS), pages 189–196, Toulouse, France, November 2010. IRIT Press.

[GCS11] Laurent GEORGE, Pierre COURBIN et Yves SOREL :
Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling.
Journal of Systems Architecture, 57(5):518–535, May 2011.

[GD97] Joël GOOSSENS et Raymond DEVILLERS :
The non-optimality of the monotonic priority assignments for hard real-time offset free systems.
Real-Time Systems, 13(2):107–126, September 1997.

[Geo08] Laurent GEORGE :
Etat de l'art sur la robustesse temporelle des systèmes temps-réel mono-processeur.
Journal Européen des Systèmes Automatisés, 42(9):1135–1160, 2008.

[GFB03] Joël GOOSSENS, Shelby Hyatt FUNK et Sanjoy K. BARUAH :
Priority-driven scheduling of periodic task systems on multiprocessors.
Real-Time Systems, 25(2-3):187–205, September 2003.

[GJ79] Michael R. GAREY et David S. JOHNSON :
Computers and Intractability : A Guide to the Theory of NP-Completeness.
1979.

[GLDN01] Paolo GAI, Giuseppe LIPARI et Marco DI NATALE :
Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip.
In Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS), pages 73–83, London, UK, December 2001. IEEE Computer Society.

[GRS96] Laurent GEORGE, Nicolas RIVIERRE et Marco SPURI :
Preemptive and non-preemptive real-time uniprocessor scheduling.
Rapport technique RR-2966, INRIA, Rocquencourt, France, September 1996.

[GSYY10] Nan GUAN, Martin STIGGE, Wang YI et Ge YU :
Fixed-priority multiprocessor scheduling with liu & layland's utilization bound.

In Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 165–174, Stockholm, Sweden, April 2010. IEEE Computer Society.

[HL94] Rhan HA et Jane W. S. LIU :
Validating timing constraints in multiprocessor and distributed real-time systems.
In Proceedings of the 14th International Conference on Distributed Computing Systems (ICDCS), pages 162–171, Pozman, Poland, June 1994. IEEE Computer Society.

[JP86] Mathai JOSEPH et Paritosh K. PANDYA :
Finding response times in a real-time system.
The Computer Journal, 29(5):390–395, 1986.

[KGV83] Scott KIRKPATRICK, C. D. GELATT et Mario P. VECCHI :
Optimization by simulated annealing.
Science, 220(4598):671–680, May 1983.

[KI09] Shinpei KATO et Yutaka ISHIKAWA :
Gang EDF scheduling of parallel task systems.
In Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS), pages 459–468, Washington, D.C., USA, December 2009. IEEE Computer Society.

[KY08] Shinpei KATO et Nobuyuki YAMASAKI :
Portioned static-priority scheduling on multiprocessors.
In Proceedings of the 22th IEEE International Parallel and Distributed Processing Symposium (IPDPS), pages 1–12, Miami, Florida, USA, April 2008. IEEE Computer Society.

[KY09] Shinpei KATO et Nobuyuki YAMASAKI :
Semi-partitioned fixed-priority scheduling on multiprocessors.
In Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), pages 23–32, San Francisco, California, USA, April 2009. IEEE Computer Society.

[KY11] Shinpei KATO et Nobuyuki YAMASAKI :
Global EDF-based scheduling with laxity-driven priority promotion.
Elsevier Journal of Systems Architecture, 57(5):498–517, May 2011.

[KYI09] Shinpei KATO, Nobuyuki YAMASAKI et Yutaka ISHIKAWA :
Semi-partitioned scheduling of sporadic task systems on multiprocessors.
In Proceedings of the 21st Euromicro Conference on Real-time Systems (ECRTS), pages 249–258, Dublin, Ireland, July 2009. IEEE Computer Society.

[LA10] Hennadiy LEONTYEV et James H. ANDERSON :

Generalized tardiness bounds for global multiprocessor scheduling.
Real-Time Systems, 44(1-3):26–71, March 2010.
10.1007/s11241-009-9089-2.

[LDG04] José M. LÓPEZ, José L. DÍAZ et Daniel F. GARCÍA :
Utilization bounds for EDF scheduling on real-time multiprocessor systems.
Real-Time Systems, 28(1):39–68, October 2004.

[LdNR09] Karthik LAKSHMANAN, Dionisio de NIZ et (Raj) Ragunathan RAJKUMAR :
Coordinated task scheduling, allocation and synchronization on multiprocessors.
In Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS), pages 469–478, Washington, D.C., USA, December 2009. IEEE Computer Society.

[Lee94] Suk Kyoon LEE :
On-line multiprocessor scheduling algorithms for real-time tasks.
In Proceedings of the IEEE Region 10's Ninth Annual International Conference (TENCON), volume 2, pages 607–611. IEEE Computer Society, August 1994.

[Leu89] Joseph Yuk-Tong LEUNG :
A new algorithm for scheduling periodic real-time tasks.
Algorithmica, 4(1-4):209–219, June 1989.

[LFPB10] Greg LEVIN, Shelby Hyatt FUNK, Ian PYE et Scott BRANDT :
DP-Fair : A simple model for understanding optimal multiprocessor scheduling.
In Proceedings of the 22nd Euromicro Conference on Real-time Systems (ECRTS), pages 3–13, Brussels, Belgium, July 2010. IEEE Computer Society.

[LGDG00] José M. LÓPEZ, M. GARCÍA, José L. DÍAZ et Daniel F. GARCÍA :
Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems.
In Proceedings of the 12th Euromicro Conference on Real-time Systems (ECRTS), pages 25–33, Stockholm , Sweden, June 2000. IEEE Computer Society.

[LKRR10] Karthik LAKSHMANAN, Shinpei KATO et Ragunathan (RAJ) RAJKUMAR :
Scheduling parallel real-time tasks on multi-core processors.
In Proceedings of the 31st IEEE Real-Time Systems Symposium (RTSS), pages 259–268, San Diego, CA, USA, November-December 2010. IEEE Computer Society.

[LL73] Chung Laung LIU et James W. LAYLAND :

Scheduling algorithms for multiprogramming in a hard-real-time environment.
Journal of the ACM, 20(1):46–61, January 1973.

[LMM98] Sylvain LAUZAC, Rami MELHEM et Daniel MOSSÉ :
An efficient RMS admission control and its application to multiprocessor scheduling.
In Proceedings of the 12th International Parallel Processing Symposium (IPPS), pages 511–518. IEEE Computer Society, March–April 1998.

[LW82] Joseph Yuk-Tong LEUNG et Jennifer WHITEHEAD :
On the complexity of fixed-priority scheduling of periodic, real-time tasks.
Performance Evaluation, 2(4):237–250, December 1982.

[Mas08] Damien MASSON :
Intégration des événements non périodiques dans les systèmes temps réel – Application à la gestion des événements dans la spécification temps réel pour Java.
Thèse de doctorat, Université Paris-Est, Marne-la-Vallée, France, December 2008.

[Mok83] Aloysius Ka-Lau MOK :
Fundamental Design Problems of Distributed Systems for Hard-Real-Time Environments.
Thèse de doctorat, Massachusetts Institute of Technology, Cambridge, MA, USA, May 1983.

[OS93] Yingfeng OH et Sang H. SON :
Tight performance bounds of heuristics for a real-time scheduling problem.
Rapport technique CS-93-24, University of Virginia, Charlottesville, VA, USA, May 1993.

[OS95] Yingfeng OH et Sang H. SON :
Allocating fixed-priority periodic tasks on multiprocessor systems.
Real-Time Systems, 9(3):207–239, November 1995.

[QFM11] Manar QAMHIEH, Frédéric FAUBERTEAU et Serge MIDONNET :
Performance analysis for segment stretch transformation of parallel real-time tasks.
In Proceedings of the 2th Junior Researcher Workshop on Real-Time Computing (JRWRTC), page 4pp, Nantes, France, September 2011.

[RR90] Ragunathan (RAJ) RAJKUMAR :
Real-time synchronization protocols for shared memory multiprocessors.

In Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS), pages 116–123, Paris, France, May-June 1990. IEEE Computer Society.

[SJNM04] Frank SINGHOFF, Legrand JÉRÔME, Laurent NANA et Lionel MARCÉ :
Cheddar : a flexible real time scheduling framework.
ACM SIGAda Ada Letters, 24(4):1–8, December 2004.

[SRL90] Lui SHA, Ragunathan RAJKUMAR et John P. LEHOCZKY :
Priority inheritance protocols : An approach to real-time synchronization.
IEEE Transactions on Computers, 39(9):1175–1185, September 1990.

[Sta88] John A. STANKOVIC :
Misconceptions about real-time computing : A serious problem for next-generation systems.
Computer, 21(10):10–19, October 1988.

[SWPT03] Christoph STEIGER, Herbert WALDER, Marco PLATZNER et Lothar THIELE :
Online scheduling and placement of real-time tasks to partially reconfigurable devices.
In Proceedings of the 30th IEEE Real-Time Systems Symposium (RTSS), pages 224–235, Cancun, Mexico, December 2003. IEEE Computer Society.

[TBW92] Ken TINDELL, Alan BURNS et Andy J. WELLINGS :
Allocating hard real-time tasks : An NP-hard problem made easy.
Real-Time Systems, 4(2):145–165, June 1992.

[WCL+07] Hsin-Wen WEI, Yi-Hsiung CHAO, Shun-Shii LIN, Kwei-Jay LIN et Wei-Kuan SHIH :
Current results on EDZL scheduling for multiprocessor real-time systems.
In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), pages 120–130, Daegu, Korea, August 2007. IEEE Computer Society.

[YSKA+08] Hamid Tabatabaee YAZDI, Vahid SALMANI, Narges KHATIB-ASTANEH, Mahdi SALMANI et Amin Milani FARD :
Exploiting laxity for heterogeneous multiprocessor real-time scheduling.
In Proceedings of the 3rt International Conference on Information and Communication Technologies : From Theory to Applications (ICTTA), pages 1–6, Damascus, Syria, April 2008. IEEE Communications Society.

[ZMM03] Dakai ZHU, Daniel MOSSE et Rami MELHEM :
Multiple-resource periodic scheduling problem : how much fairness is necessary ?
In Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS), pages 142–151, Cancun, Mexico, December 2003. IEEE Computer Society.

GLOSSAIRE

A

Activation Date à laquelle une instance est prête à pouvoir démarrer son exécution. 4

E

Échéance 4
- **absolue** Date à laquelle une instance d'une tâche doit avoir terminé son exécution. 4
- **arbitraire** Échéance non contrainte par la période. 5
- **contrainte** Échéance inférieure ou égale à la période. 5, 29, 45, 57
- **implicite** Échéance strictement égale à la période. 5
- **relative** Échéance relative à la date d'activation d'une instance de la tâche. 4

I

Instance Occurrence d'une tâche temps réel. 4

M

Migration Déplacement d'une tâche ou d'une instance de cette tâche d'un processeur vers un autre. 13, 57

O

Ordonnancement Mécanisme consistant à décider dans quel ordre exécuter les tâches. 2
- **en ligne** Ordonnancement pour lequel les décisions d'ordonnancement sont prises durant l'ordonnancement. 8
- **hors-ligne** Ordonnancement pour lequel les décisions d'ordonnancement sont connues avant le démarrage de l'ordonnanceur. 8
- **non oisif** Ordonnancement pour lequel aucun processeur ne peut rester inactif alor que des instances sont actives. 9

non préemptif Ordonnancement pour lequel l'exécution d'une instance ne peut pas être suspendue. 8

oisif Ordonnancement pour lequel au moins un processeur peut rester inactif alors que des instances sont actives. 9

préemptif Ordonnancement pour lequel l'exécution d'une instance peut être suspendue. 8

P

Période Intervalle fixé de temps séparant l'activation de deux instances d'une tâche temps réel. 3

d'inter-arrivé minimale Intervalle minimum de temps séparant l'activation de deux instances d'une tâche temps réel. 3

S

Sysème

critique Système dont la défaillance peut avoir des conséquences graves. 1

temps réel Un système en charge de l'exécution d'un ensemble de tâches temps réel sur une plate-forme donnée. 1, 2

multiprocesseur système dont la plate-forme est constituée de plusieurs processeurs. 2

souple Système pour lequel le dépassement d'un certain nombre d'échéances est autorisé dans la mesure où une certaine qualité peut être garantie. 3

strict Système pour lequel aucun dépassement d'échéance n'est autorisé. 3

T

Tâche

périodique Tâche dont la période est constante. 3, 57

sporadique Tâche pour laquelle seule une durée minimale entre deux activations d'instances est connue. 3, 29, 45, 67

temps réel Tâche dont toutes les instances doivent avoir terminé leur exécution avec une échéance fixée. 2

Temps de réponse Durée entre l'instant d'activation et l'instant de terminaison. 6, 9

ACRONYMES

A

AF (AF) *Allowance-Fit. 38*
AF^C *(AF^C)* Allowance-Fit-WCET. 37, 38, 40–43
AF^f (AF^f) *Allowance-Fit-Frequency. 37, 38, 42, 43*
ASMP *(ASMP) multitraitement asymétrique (*Asymmetric multiprocessing*). 7*
AWF *(AWF)* Almost-Worst-Fit. 34, 38

B

BF (BF) *Best-Fit. 32, 33, 36–38, 50, 79*

D

***D*Π** *(D*Π*) affectation dynamique des instances aux processeurs (*Dynamic Processor*). 14, 15, 57, 80, 81*
DD *(DD) échéances décroissantes (*Decreasing Deadline*). 36, 79*
DM *(DM)* Deadline-Monotonic. 10, 12, 37, 38, 43, 70, 79
DP (DP) priorités dynamiques (*Dynamic Priority*). 11, 14, 15, 83
DU (DU) utilisation décroissante (*Decreasing Utilization*). 36–38, 40, 41, 49, 50, 79
DW (DW) WCET décroissant (*Decreasing WCET*). 36

E

EDCL (EDCL) *Earliest Deadline until Critical Laxity. 15*
EDF *(EDF)* Earliest Deadline First. 10, 11, 59, 66, 83
EDZL (EDZL) *Earliest Deadline until Zero Laxity. 15*

F

F-AWF *(F-AWF)* Fixed-Almost-Worst-Fit. 35, 38, 40, 42
FF (FF) *First-Fit. 32, 35, 37, 38, 79*
FIFO *(FIFO)* First In First Out. 49, 106, 107

FJΠ (FJΠ) affectation fixe des instances aux processeurs (*Fixed Job Processor*). 14, 15, 18, 57–59, 64, 70, 80
FJP (FJP) priorités fixes au niveau des instances (*Fixed Job Priority*). 10, 11, 14, 15, 83
FMLP (FMLP) *Flexible Multiprocessor Locking Protocol. 48–51, 53, 55*
FPZL *(FPZL)* Fixed Priority until Zero Laxity. 15
FTΠ (FTΠ) affectation fixe des tâches aux processeurs (*Fixed Task Processor*). 14–16, 29, 43, 45, 55, 84
FTP (FTP) priorités fixes au niveau des tâches (*Fixed Task Priority*). 10–12, 14, 15, 17–20, 29, 43, 48, 55, 57–59, 61, 64, 67, 70, 80, 81, 83, 84
F-WF (F-WF) *Fixed-Worst-Fit. 35, 38, 40–42*

I

ID *(ID) échéances croissantes (*Increasing Deadline*). 36, 37, 79*
IL *(IL) laxité croissante (*Increasing Laxity*). 40, 41, 43*
IP *(IP) périodes croissantes (*Increasing Period*). 36, 37, 79*
IU *(IU) utilisation croissante (*Increasing Utilization*). 36*
IW *(IW) WCET croissants (*Increasing WCET*). 36*

L

LF *(LF)* Last-Fit. 32, 38
LLF (LLF) *Least Laxity First. 11, 59, 83*

M

MPCP *(MPCP)* Multiprocessor Priority Ceiling Protocol. 47–51, 55
MSRP (MSRP) *Multiprocessor Stack Resource Policy. 48, 50, 51, 55*

N

NF *(NF)* Next-Fit. 32, 37, 38, 79
NUMA (NUMA) accès mémoire non uniforme (*Non-Uniform Memory Access*). 7

O

OPA (OPA) attribution des priorités optimal (*Optimal Priority Assignment*). 10, 12

P

PCP (PCP) *Priority Ceiling Protocol. 47, 48*

R

RBF *(RBF)* Request Bound Function. 79
RM (RM) *Rate-Monotonic. 10, 12, 37*
RPSA *(RPSA)* Robust Partitioning based on Simulated Annealing. 50, 51, 53, 54, 79

RTMSim (RTMSim) Real-Time Multiprocessor SIMulator. 75–77, 81
RTSJ (RTSJ) spécification temps réel pour Java (*Real-time Specification for Java*). 81

S

SMP (SMP) multitraitement symétrique (*Symmetric multiprocessing*). 7
SPA (SPA) *Synchronization-Aware Partitioning Algorithm. 49, 53, 54, 79*
SRP *(SRP)* Stack Resource Policy. 48

U

UMA (UMA) accès mémoire uniforme (*Uniform Memory Access*). 7

W

WCET (WCET) durée d'exécution pire cas (*Worst Case Execution Time*). 5
WF (WF) *Worst-Fit. 33–35, 38, 50*

X

XML *(XML)* eXtensible Markup Language. 76

ANNEXE A

PROTOCOLES DE SYNCHRONISATION

A.1 MPCP

Notation	Définition
J_i	Une instance de τ_i
π_j	Le processeur d'indice j
$\mathcal{R}_k$	La ressource locale d'indice k
$\mathcal{R}_{Gk}$	La ressource globale d'indice k
$p(J_i)$	La priorité de J_i
$\overline{p}(\mathcal{R}_k)$	La priorité plafond de $\mathcal{R}_k$
$\overline{p}(t)$	La priorité plafond du système à l'instant t
$\overline{p}(t, \pi_j)$	La priorité plafond de π_j à l'instant t

TABLE A.1 – Notations pour MPCP.

Dans la figure A.1, nous représentons un exemple donné dans [RR90] correspondant à l'ordonnancement de 7 instances ($\{J_1, \ldots, J_7\}$) réparties sur 3 processeurs ($\{\pi_1, \pi_2, \pi_3\}$), partageant 3 ressources locales ($\{\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3\}$) et 2 ressources globales ($\{\mathcal{R}_{G1}, \mathcal{R}_{G2}\}$). La plus haute priorité du système est définie par $p(J_1) + 1$.

- à $t = 0$, J_3 et J_6 sont les seules instances actives sur leur processeur respectif et démarrent donc leur exécution ;
- à $t = 1$, J_3 utilise $\mathcal{R}_{G1}$ et est exécutée à la priorité $p(\mathcal{R}_{G1}) = p_G + p(J_1)$ (car J_1 est l'instance la plus prioritaire pouvant utiliser $\mathcal{R}_{G1}$). J_6 utilise R_2 car la ressource est libre ;
- à $t = 2$, J_1 est activée mais ne peut pas démarrer car J_3 est exécutée à la priorité $p(\mathcal{R}_{G1})$. J_4 est activée et préempte J_6 moins prioritaire ;
- à $t = 3$, J_3 libère $\mathcal{R}_{G1}$ et redevient moins prioritaire que J_1 qui la préempte pour démarrer son exécution. J_4 veut utiliser R_2, mais sa priorité n'est pas strictement supérieure à $p(R_2) = p(J_4)$;

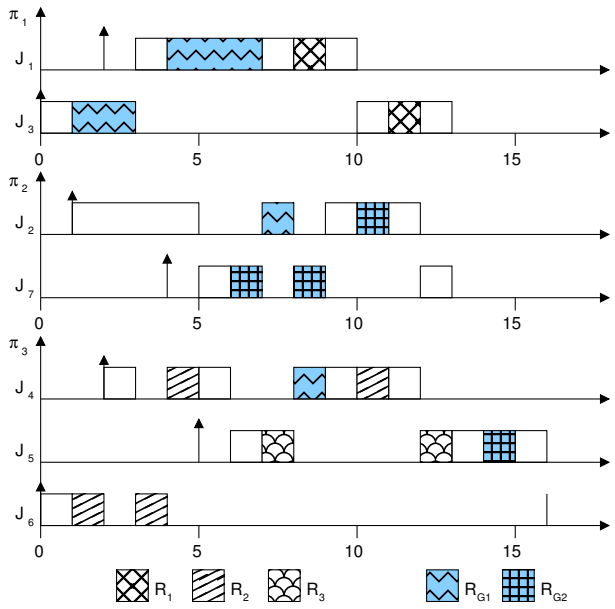


FIGURE A.1 – Exemple d'ordonnancement avec MPCP.

- à $t = 4$, J_1 utilise $\mathcal{R}_{G1}$. J_7 est activée mais est moins prioritaire que J_2. J_6 libère R_2 et est donc préemptée par J_4 qui utilise alors R_2 ;
- à $t = 5$, J_2 veut utiliser $\mathcal{R}_{G1}$ mais celle-ci est déjà utilisée par J_1. J_2 est alors bloquée. J_5 est activée mais est moins prioritaire que J_4 ;
- à $t = 6$, J_7 utilise $\mathcal{R}_{G2}$ et est donc exécutée à la priorité $p(\mathcal{R}_{G2}) = p_G + p(J_2)$. J_4 veut utiliser $\mathcal{R}_{G1}$ mais celle-ci est déjà utilisée par J_1. J_4 est alors bloquée. J_5 peut donc démarrer son exécution ;
- à $t = 7$, J_1 libère $\mathcal{R}_{G1}$. J_2 peut donc l'utiliser car c'est l'instance la plus prioritaire en attente de cette ressource et $p(\mathcal{R}_{G1}) > p(\mathcal{R}_{G2}$. J_5 utilise la ressource $\mathcal{R}_3$;
- à $t = 8$, J_1 utilise $\mathcal{R}_1$. J_2 libère $\mathcal{R}_{G1}$ et est donc préemptée par J_7 qui s'exécute à la priorité $p(\mathcal{R}_{G2}$. J_4 peut alors préempter J_5 pour utiliser $\mathcal{R}_{G1}$;
- à $t = 9$, J_7 libère $\mathcal{R}_{G2}$ et retrouve sa priorité initiale. Elle est donc préemptée ;
- à $t = 10$, J_1 termine son exécution et permet ainsi à J_3 de continuer la sienne. J_4 utilise $\mathcal{R}_2$ car sa priorité plafond est supérieure à celle de $\mathcal{R}_3$ en cours d'utilisation ;
- à $t = 11$, J_3 utilise $\mathcal{R}_{G1}$ car cette ressource est libre ;
- à $t = 12$, J_2 termine son exécution et permet ainsi à J_7 de continuer la sienne. J_5 peut reprendre l'utilisation de $\mathcal{R}_{G2}$ après la terminaison de J_4 ;
- à partir de $t = 13$, J_5 est l'instance restante la plus prioritaire et termine donc son exécution à $t = 16$, permettant ainsi à J_6 de terminer à son tour.

Notation	Définition
J_i	Une instance de τ_i
π_j	Le processeur d'indice j
$\mathcal{R}_k$	La ressource locale d'indice k
$\mathcal{R}_{Gk}$	La ressource globale d'indice k
$\rho(J_1)$	Le niveau de préemption de J_1
$\rho(\mathcal{R}_k)$	Le niveau de préemption de $\mathcal{R}_k$
$\overline{\rho}(t)$	Le niveau de préemption plafond du système à l'instant t
$\overline{\rho}(t, \pi_j)$	Le niveau de préemption plafond du système à l'instant t sur π_j

TABLE A.2 – Notations pour MSRP.

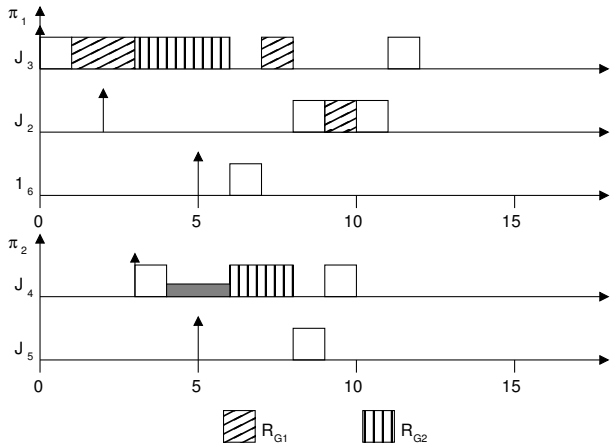


FIGURE A.2 – Exemple d'ordonnancement avec MSRP

A.2 MSRP

Dans la figure A.2, nous représentons un exemple donné dans [GLDN01] correspondant à l'ordonnancement de 4 instances ($\{J_1, \ldots, J_5\}$) réparties sur 2 processeurs ($\{\pi_1, \pi_2\}$) et partageant 2 ressources globales ($\{\mathcal{R}_{G1}, \mathcal{R}_{G2}\}$). Les niveaux de préemptions sont $\rho(J_1) = 3$, $\rho(J_2) = 2$, $\rho(J_3) = 1$, $\rho(J_4) = 1$ et $\rho(J_5) = 2$.

- à $t = 2$, J_2 est bloquée car son niveau de préemption n'est pas strictement supérieur au plafond $\overline{\rho}(t, \pi_1)$;
- à $t = 3$, J_3 utilise $\mathcal{R}_{G2}$ et élève $\overline{\rho}(t, \pi_1)$ à 3 ;
- à $t = 4$, J_4 veut utiliser $\mathcal{R}_{G2}$ qui n'est pas libre. Il élève donc $\overline{\rho}(t, \pi_2)$ à 2 et se met en attente active ;
- à $t = 5$, J_1 et J_5 sont bloqués car les plafonds de préemption de π_1 et π_2 sont au maximum ;
- à $t = 6$, J_3 libère $\mathcal{R}_{G2}$ et J_4 peut alors l'utiliser. $\overline{\rho}(t, \pi_1)$ est rétabli à 2 et J_1 peut donc préempter J_3.

A.3 FMLP

A.3.1 Exemple

Notation	Définition
J_i	Une instance de τ_i
π_j	Le processeur d'indice j
$\mathcal{R}_k$	La ressource locale d'indice k
$\mathcal{R}_{Gk}$	La ressource globale d'indice k
$\mathcal{R}^l$	Une ressource longue
$\mathcal{R}^s$	Une ressource courte

TABLE A.3 – Notations pour FMLP.

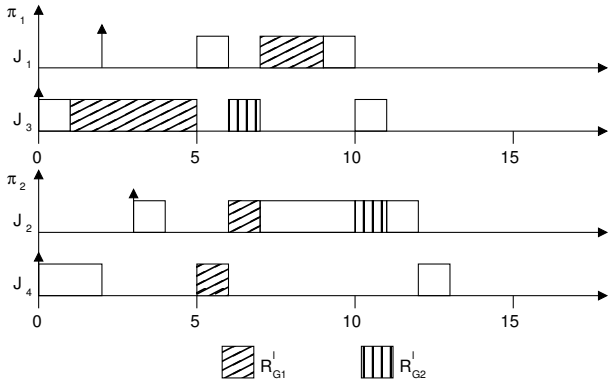


FIGURE A.3 – Exemple d'ordonnancement avec FMLP (ressources longues).

Dans la figures A.3, nous représentons un exemple donné dans [BA08] correspondant à l'ordonnancement de 4 instances ($\{J_1, \ldots, J_4\}$) réparties sur 2 processeurs ($\{\pi_1, \pi_2\}$) et partageant 2 ressources globales longues ($\{\mathcal{R}^l_{G1}, \mathcal{R}^l_{G2}\}$).

- à $t = 0$, J_3 et J_4 sont les seules instances actives sur leur processeur respectif et démarrent donc leur exécution ;
- à $t = 1$, J_3 utilise $\mathcal{R}^l_{G1}$ car celle-ci est libre ;
- à $t = 2$, J_1 est activée mais ne peut pas démarrée car, bien que plus prioritaire que J_3, cette dernière est exécutée de manière non préemptive. J_4 veut utiliser $\mathcal{R}^l_{G1}$ mais celle-ci n'est pas libre. J_4 est donc suspendue en attente de la libération de $\mathcal{R}^l_{G1}$;
- à $t = 4$, J_2 est à son tour suspendue en attente de $\mathcal{R}^l_{G1}$;
- à $t = 5$, J_3 libère $\mathcal{R}^l_{G1}$, la rendant ainsi disponible pour J_4, qui est la première instance à avoir été mise en attente pour cette ressource. J_1 démarre son exécution en préemptant J_3 ;
- à $t = 6$, J_1 est suspendue en attente de $\mathcal{R}^l_{G1}$. J_4 libère $\mathcal{R}^l_{G1}$, la rendant disponible pour J_2 (J_1 est plus prioritaire que J_2 mais J_2 est avant J_1 dans la FIFO). J_3 utilise $\mathcal{R}^l_{G2}$ car celle-ci est libre ;

- à $t = 7$, J_3 libère $\mathcal{R}^l_{G2}$, rendant ainsi π_1 disponible et J_2 libère $\mathcal{R}^l_{G1}$. J_1 peut donc utiliser $\mathcal{R}^l_{G1}$. À partir de cet instant, il n'y a plus de requêtes concurrentes pour les ressources ;
- à $t = 13$, toutes les instances ont terminé leur exécution.

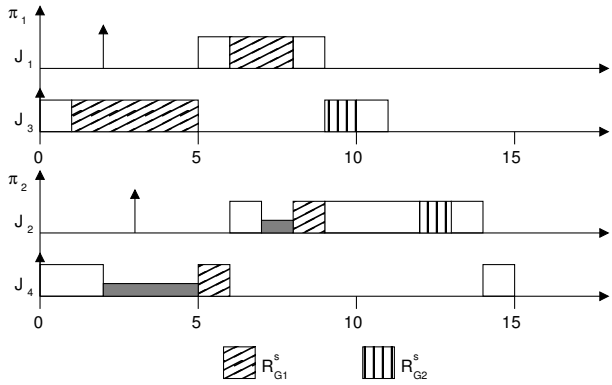


FIGURE A.4 – Exemple d'ordonnancement avec FMLP (ressources courtes).

Dans la figures A.4, nous représentons l'ordonnancement de 4 instances ($\{J_1, \ldots, J_4\}$) réparties sur 2 processeurs ($\{\pi_1, \pi_2\}$) et partageant 2 ressources globales courtes ($\{\mathcal{R}^s_{G1}, \mathcal{R}^s_{G2}\}$). Les différences avec l'ordonnancement de la figure A.3 sont les suivantes :

- à $t = 2$, J_4 ne peut pas utiliser $\mathcal{R}^s_{G1}$, elle est donc placée en attente active dans une FIFO ;
- à $t = 7$, J_2 ne peut pas utiliser $\mathcal{R}^s_{G1}$, elle est donc placée en attente active dans une FIFO.

A.3.2 Pire temps de blocage

Pour pouvoir calculer le pire temps de réponse d'une tâche τ_i, il est nécessaire de connaître le pire temps de blocage B_i qu'elle peut subir. Ce pire temps de blocage B_i se décompose en 5 facteurs :

- le blocage local court noté AB_i (pour *Arrival Blocking*),
- le blocage local long noté BB_i (pour *Boost Blocking*),
- le blocage global court noté SB_i (pour *Short Blocking*),
- le blocage global long noté LB_i (pour *Long Blocking*),
- le blocage différé long DB_i.

Blocage local court Ce facteur inclut le temps de blocage induit par une tâche τ_j moins prioritaire que τ_i lorsque τ_j est en attente sur une section critique courte et que τ_i et τ_j sont ordonnancées sur le même processeur. τ_j est donc en train d'être exécutée de manière non préemptive et retarde le démarrage de τ_i. Le pire temps de blocage

pour ce facteur est donné par :

$$AB_i = \sum_{l=1}^{\min(na,nb)} |\mathcal{R}_l|$$

où na est le nombre possible d'arrivées de τ_i et où nb est le nombre de blocages que τ_i peut subir. Le nombre d'arrivées possibles de la tâche τ_i est défini par :

$$na = 1 + |RL(\tau_i)|$$

En effet, τ_i peut être démarrée après son activation et dans le pire cas après avoir été bloquée pour chaque section critique longue. Le nombre de blocage nb que τ_i peut subir correspond à la taille de l'ensemble abr et est défini par :

$$nb = |abr(\tau_i)|$$

où l'ensemble abr représente l'ensemble des sections critiques courtes qui peuvent être partagées avec des tâches de priorité inférieure à celle de τ_i. Cet ensemble est défini par :

$$abr(\tau_i) = \bigcup_{\substack{\tau_j \in lp(\tau_i) \\ \pi(\tau_i)=\pi(\tau_j)}} wcsx(\tau_i)$$

où $wcsx(\tau_i)$ est le nombre de sections critiques courtes qui peuvent être exécutées par d'autres tâches que τ_i. $wcsx$ est défini par :

$$wcsx(\tau_i) = \left\{ (\mathcal{R}, k) | \mathcal{R} \in RC(\tau_i) \wedge k \in \left\{ 1, \ldots, \left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right\} \right\}$$

Le facteur AB_i est égal à la somme de $n = \min(na, nb)$ durées de sections critiques. Ces n sections critiques sont les n premiers éléments de l'ensemble abr trié par ordre croissant des durées des sections critiques.

Blocage local long Ce facteur inclut le temps de blocage induit par une tâche τ_j moins prioritaire que τ_i dont la priorité a été élevée parce que celle-ci exécute une section critique longue ordonnancée sur le même processeur que τ_i. Le pire temps de blocage pour ce facteur est donné par :

$$BB_i = \sum_{\substack{\tau_j \in lp(\tau_i) \\ \pi(\tau_i)=\pi(\tau_j)}} bbt(\tau_j, \tau_i)$$

où la fonction bbt donne la durée pire cas du blocage induit par une tâche de plus faible priorité que τ_i. Cette fonction est définie par :

$$bbt(\tau_j, \tau_i) = \sum_{l=1}^{\min(na,nb)} |\mathcal{R}_l|$$

où na correspond toujours au nombre d'arrivées possibles de τ_i défini par :

$$na = 1 + |RL(\tau_i)|$$

et nb le nombre de blocage que τ_i peut subir. Ce nombre de blocage est borné par la taille de l'ensemble des sections critiques qui peuvent être exécutées par une tâche de priorité inférieure à τ_i et est défini par :

$$nb = |wclx(\tau_i)|$$

où $wclx(\tau_i)$ est donné par :

$$wclx(\tau_i) = \left\{ (\mathcal{R}, k) | \mathcal{R} \in RL(\tau_i) \wedge k \in \left\{ 1, \ldots, \left\lceil \frac{T_i}{T_j} \right\rceil + 1 \right\} \right\}$$

Les $n = \min(na, nb)$ durées de sections critiques composant la somme dans la fonction bbt sont les n premiers éléments de l'ensemble $wclx$ trié dans l'ordre croissant de durée des sections critiques.

Blocage global court Ce facteur inclut le temps de blocage induit par les tâches partageant des ressources courtes avec τ_i et ordonnancées sur des processeurs différents de celui sur lequel est ordonnancée τ_i. Le pire temps de blocage pour ce facteur est donné par :

$$SB_i = \sum_{\mathcal{R} \in RC(\tau_i)} sbr(\tau_i, \mathcal{R})$$

où la fonction sbr donne la pire durée de blocage que peut subir la tâche τ_i en attendant de pouvoir exécuter la section critique $\mathcal{R}$. Cette fonction est définie par :

$$sbr(\tau_i, \mathcal{R}) = \sum_{\substack{\pi_k \in \Pi \\ \pi_k \neq \pi(\tau_i)}} \max(\mathcal{R} | \pi(\tau_j) = \pi_k \wedge \mathcal{R} \in RC(\tau_i) \wedge \mathcal{R} \in RC(\tau_j))$$

Blocage global long Ce facteur inclut le temps de blocage induit lorsque la tâche τ_i tente d'exécuter une section critique longue. Deux cas peuvent se présenter en considérant que τ_i est bloquée par une tâche τ_j avec laquelle elle partage une ressource et qui est ordonnancée sur un processeur différent de celui sur lequel est ordonnancé τ_i. Soit τ_j est bloquée parce qu'une tâche s'exécute de manière non préemptive en attente

d'une ressource courte. Soit τ_j est bloquée parce qu'une tâche exécute une section critique longue et a donc sa priorité élevée. Le pire temps de blocage pour ce facteur est donc donné par :

$$LB_i = rbs(\tau_i) + rbl(\tau_i)$$

La fonction rbs donne le pire temps de blocage que peut subir τ_i par un blocage transitif distant sur une ressource courte. Cette fonction est définie par :

$$rbs(\tau_i) = \sum_{\substack{\pi_k \in \Pi \\ \pi_k \neq \pi(\tau_i)}} \sum_{l=1}^{\min(nb,ns)} |\mathcal{R}_l|$$

où nb est le nombre de fois où τ_i est directement bloquée par une tâche qui exécute une section critique longue sur le processeur π_k. nb est défini par :

$$nb = \sum_{\tau_j \in \pi(\tau_i)} \sum_{r \in RL(\tau_i)} ntg(\tau_j, \tau_i, r)$$

où la fonction ntg est donnée par :

$$ntg(\tau_j, \tau_i, r) = \min\left(|\{\mathcal{R}|\mathcal{R} \in RL(\tau_i)\}|, |\{\mathcal{R}|\mathcal{R} \in RL(\tau_j)\}| \cdot \left(\left\lceil \frac{T_i}{T_j} \right\rceil + 1\right)\right)$$

et où ns est le nombre de sections critiques courtes qui peuvent être exécutées par des tâches τ_j. ns est défini par :

$$ns = \bigcup_{\tau_j \in \pi(\tau_i)} wcsx(\tau_i)$$

Pour rappel, $wcsx$ est défini par :

$$wcsx(\tau_i) = \left\{(\mathcal{R}, k)|\mathcal{R} \in RC(\tau_i) \wedge k \in \left\{1, \ldots, \left\lceil \frac{T_i}{T_j} \right\rceil + 1\right\}\right\}$$

Les $n = \min(nb, ns)$ durées d'utilisation des sections critiques composant la double somme de sbr sont les n premiers éléments de l'ensemble $wcsx$ trié dans l'ordre croissant des durées des sections critiques.

La fonction rbl donne le pire temps de blocage que peut subir τ_i par un blocage transitif distant sur une ressource longue. Cette fonction est définie par :

$$rbl(\tau_i) = \sum_{\substack{\pi_k \in \Pi \\ \pi_k \neq \pi(\tau_i)}} \sum_{\tau_j \in \pi_k} \sum_{l=1}^{\min(nb,ns)} |\mathcal{R}_l|$$

Cette fois, nb est défini par :

$$nb = \sum_{\tau_x \in \pi(\tau_j)} \sum_{r \in RL(\tau_i)} ntg(\tau_x, \tau_i, r)$$

et ns est défini par :

$$ns = |wclx(\tau_i)|$$

Les $n = \min(nb, ns)$ durées d'utilisation des sections critiques composant la triple somme de sbr sont les n premiers éléments de l'ensemble $wclx$ triés dans l'ordre croissant des durées des sections critiques.

Blocage différé Ce facteur inclut le temps de blocage induit lorsque une tâche de priorité supérieure à celle de τ_i a son exécution retardée à cause d'un blocage sur une section critique. Ce retard engendre une interférence accrue sur τ_i. Le pire temps de blocage pour ce facteur est donné par :

$$DB_i = \sum_{\substack{\tau_j \in hp(\tau_i) \\ \pi(\tau_j) = \pi(\tau_i)}} \min(C_j, LB_j)$$

Blocage total Le pire temps de blocage B_i pour une tâche τ_i est donc donné par la somme de tous ces facteurs :

$$B_i = AB_i + BB_i + SB_i + LB_i + DB_i$$