Contribution à l'analyse d'ordonnançabilité des applications temps-réel multiprocesseurs

Sadouanouan Malo

▶ To cite this version:

Sadouanouan Malo. Contribution à l'analyse d'ordonnançabilité des applications temps-réel multiprocesseurs. Sciences de l'ingénieur [physics]. ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique - Poitiers, 2010. Français. NNT : . tel-00554234

HAL Id: tel-00554234
https://theses.hal.science/tel-00554234

Submitted on 10 Jan 2011

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.

THESE

Pour l'obtention du Grade de

DOCTEUR DE L'ECOLE NATIONALE SUPERIEURE DE MECANIQUE ET D'AEROTECHNIQUE

(Diplôme Nationale – Arrêté du 7 Août 2006)

Ecole Doctorale : Sciences et Ingénierie pour l'Information

Secteur de Recherche : Informatique et Application

Présenté par :

Sadouanouan MALO

****

Contribution à l'analyse d'ordonnançabilité des applications temps-réel multiprocesseurs

****

Directeur de Thèse : Annie CHOQUET-GENIET

****

Soutenue le 9 Décembre 2010
devant la Commission d'Examen

****

JURY

Guy VIDAL-NAQUET	Professeur, Supélec	Rapporteur
Patrick MARTINEAU	Professeur, Polytech'Tours, LI	Rapporteur
Annie CHOQUET-GENIET	Professeur, ENSMA, LISI	Examinateur
Jean-Philippe BABAU	Professeur, Université de Brest	Examinateur
Dominique GENIET	Maître de Conférences, ENSMA, LISI	Examinateur
Gaëlle LARGETEAU-SKAPIN	Maître de Conférences, XLIM-SIC	Examinateur

REMERCIEMENTS

J'exprime mes plus vifs remerciements à Annie Choquet-Geniet pour son encadrement et les conseils inestimables qu'elle m'a donnés tout au long de mes travaux de thèse. C'est l'occasion pour moi de lui exprimer toute ma gratitude pour les connaissances acquises auprès d'elle. Sa rigueur scientifique a été déterminante pour ma formation de chercheur. Encore une fois merci.

Je présente mes remerciements au professeur Yamine Aït-Ameur pour m'avoir accepté dans le laboratoire qu'il dirige et pour avoir appuyé ma demande de financement pour la poursuite de ma thèse. Je le remercie aussi pour l'ensemble des moyens mis à ma disposition pour le déroulement de cette thèse.

Je tiens à remercier Dominique Geniet grâce à qui j'ai intégré l'équipe temps réel du LISI. Je lui exprime toute ma reconnaissance pour les conseils et le soutien tant sur le plan administratif que scientifique qu'il m'a apporté depuis mon année de DEA à l'Université Polytechnique de Bobo-Dioulasso. Du fond du cœur, merci.

J'adresse mes remerciements aux professeurs Guy Vidal-Naquet et Patrick Martineau qui ont bien voulu consacrer leur temps à la relecture et à la correction, en tant que rapporteurs de ma thèse. Je remercie par la même occasion Jean-Philippe Babau, Dominique Geniet et Gaëlle Largeteau-Skapin pour l'honneur qu'ils m'ont fait en acceptant d'être examinateurs de mon jury.

Je remercie aussi tous les membres de l'équipe temps-réel pour les remarques et discussions fructueuses qui m'ont permis d'améliorer la qualité de ce document.

Je tiens à remercier également tous les membres du LISI pour leur sympathie et les bons moments passés ensemble.

Mes remerciements s'adressent aussi au professeur Théodore M. Y. Tapsoba de l'Université Polytechnique de Bobo-Dioulasso pour avoir appuyé ma demande de bourse.

Enfin, je remercie l'Agence Universitaire pour la Francophonie pour m'avoir donné l'opportunité de poursuivre ces travaux de thèse en m'attribuant une bourse.

TABLE DES MATIÈRES

REMERCIEMENTS . . . . **iii**

TABLE DES MATIÈRES . . . . **iv**

AVANT-PROPOS . . . . **ix**

NOTATIONS . . . . **xi**

INTRODUCTION GÉNÉRALE **1**

0.1 Les systèmes temps-réel 1

0.2 Le contexte d'exécution des systèmes temps-réel 3

0.2.1 L'exécutif temps-réel 3

0.2.2 L'architecture matérielle 5

0.3 Les applications temps-réel 6

0.4 Le problème d'ordonnancement des applications temps-réel 8

0.4.1 Les ordonnancements en-ligne 11

0.4.2 Les ordonnancements hors-ligne 12

0.5 Les problèmes envisagés 13

0.5.1 Le problème de la cyclicité 13

0.5.2 La gestion des pannes 15

0.5.3 Etude des ordonnancements P-équitables dans un contexte plus large 16

0.5.4 Adjonction de nouvelles tâches 17

0.6 Organisation du document 18

I Etat de l'art **19**

CHAPITRE 1 : ORDONNANCEMENT MULTIPROCESSEUR **21**

1.1 Contexte d'exécution 21

1.1.1 Contexte matériel 21
1.1.2 Le modèle de tâche 22
1.2 Ordonnancement global versus partitionné 23
1.2.1 Ordonnancement global 24
1.2.2 Ordonnancement partitionné 29
1.2.3 Comparaison 30
1.3 Formalisation d'un ordonnancement 31
1.3.1 Instance de tâche 31
1.3.2 Ordonnancement 33
1.4 Les ordonnancements P-équitables 34
1.4.1 Principe général 34
1.4.2 Mise en oeuvre : les sous-tâches 37
1.4.3 Les algorithmes P-équitables 39
1.4.4 L'algorithme PF 39
1.4.5 L'algorithme PD 40
1.4.6 L'algorithme PD^2 42
1.5 Conclusion 43

CHAPITRE 2 : ORDONNANCEMENT HORS-LIGNE 44
2.1 Principe des approches à base de modèle 44
2.1.1 Schéma général 44
2.1.2 Approches présentes dans la littérature 45
2.2 Les approches à base de réseaux de Petri 46
2.2.1 Les réseaux de Petri 46
2.2.2 Modélisation d'une application temps-réel à base de réseaux de Petri : sémantique et prise en compte des contraintes temporelles 50
2.2.3 Analyse du système 55
2.3 Conclusion 61

II Contribution 62

CHAPITRE 3 : ETUDE DU PROBLÈME DE LA CYCLICITÉ DES ORDONNANCEMENTS MULTIPROCESSEURS 66
3.1 Le problème de la cyclicité dans un contexte d'ordonnancement multiprocesseur . . . 67
3.2 Notion de temps creux cycliques et acycliques . . . 68
3.3 Etude de la cyclicité pour la classe des ordonnancements monotones . . 72
3.3.1 Temps creux complet - temps creux partiel . . . 73
3.3.2 Cas où le dernier temps creux acyclique est complet . . . 74
3.3.3 Cas où le dernier temps creux acyclique est partiel . . . 76
3.4 La classe des ordonnancements monotones . . . 78
3.4.1 Ordonnancements à priorités fixes . . . 78
3.4.2 Cas des ordonnancements EDF . . . 81
3.4.3 Cas des ordonnancements LLF . . . 82
3.4.4 Cas des ordonnancements P-équitables . . . 83
3.5 Date du dernier temps creux acycliques . . . 87
3.6 Conclusion . . . 90

CHAPITRE 4 : RECONFIGURATION EN CAS DE PANNE PROCESSEUR . . . 92
4.1 Périmètre de l'étude . . . 92
4.1.1 Nos hypothèses . . . 92
4.1.2 Les différents cas de figures . . . 93
4.1.3 Incidence sur les paramètres des tâches . . . 95
4.2 Quelques résultats concernant la reprise sous ordonnancement EDF . . 96
4.2.1 Cas S1 et S2-P3 . . . 98
4.2.2 Cas S2 - P1 et P2 . . . 103
4.3 Quelques résultats concernant la reprise sous ordonnancements équitables 105
4.3.1 Cas S1 et S2-P3 . . . 105
4.3.2 S2 - P1 et P2 . . . 105

Conclusion 106

CHAPITRE 5 : ETUDE DES ORDONNANCEMENTS P-ÉQUITABLES POUR LES TÂCHES À DÉPARTS DIFFÉRÉS ET À ÉCHÉANCES CONTRAINTES 107

5.1 Motivation et état de l'art 107

5.2 Condition suffisante d'ordonnançabilité 108

5.2.1 Formulation de l'équité dans ce contexte 108

5.2.2 Preuve du théorème 109

5.2.3 Mise en œuvre 116

5.3 Pertinence de la condition et application 117

5.3.1 Pertinence de la condition 117

5.3.2 Application 120

Conclusion 122

CHAPITRE 6 : GESTION DES TÂCHES APÉRIODIQUES 123

6.1 Le problème 123

6.1.1 Hypothèses sur les tâches 123

6.1.2 Méthodes classiques d'ordonnancement des tâches apériodiques 124

6.2 Principe général - Gestion des temps creux 126

6.3 Le protocole d'acceptation 128

6.4 Analyse de performance 130

6.5 Généralisation 133

6.5.1 Compléments sur l'approche à base de réseaux de Petri 133

6.5.2 Extraction des séquences avec temps creux équitablement répartis 134

6.5.3 Autres critères 138

Conclusion 141

CHAPITRE 7 : CONCLUSION ET PERSPECTIVES 142

7.1 Conclusion 142

7.2 Perspectives 143

7.2.1 Les travaux en cours 144
7.2.2 Études à venir 150

LISTE DES TABLEAUX **152**

LISTE DES FIGURES **153**

BIBLIOGRAPHIE **156**

AVANT-PROPOS

Cette préface présente brièvement l'intérêt de la théorie de l'ordonnancement pour les systèmes temps-réel multiprocesseurs. Elle donne les motivations de cette thèse et présente sa structuration.

Les applications temps réel, le plus souvent dédiées au contrôle de procédé, sont soumises à des contraintes temporelles strictes, destinées à garantir la sécurité et la cohérence du procédé contrôlé. Les applications temps-réel étant des applications multitâches, elles doivent être ordonnancées, le critère sine qua none de qualité de la stratégie d'ordonnancement étant la garantie du respect des contraintes temporelles.

Ces applications sont de plus en plus déployées sur des architectures multiprocesseurs. Le problème de l'ordonnancement doit donc être posé dans ce contexte, où de nombreux problèmes doivent encore être abordés. Notons tout d'abord que dans le cas multiprocesseur, il n'existe pas d'ordonnancement en ligne optimal dans le cas général, le problème de l'ordonnancement est NP-complet, et des anomalies d'ordonnancement apparaissent même lorsque l'on ne considère que des tâches indépendantes (une durée d'exécution plus courte que prévue peut provoquer une faute temporelle). Les ordonnancements P-équitables ainsi que les stratégies d'ordonnancement hors-ligne sont donc très prometteuses dans ce contexte. De même que pour le cas monoprocesseur, dans le cadre de l'utilisation de stratégies d'ordonnancement hors-ligne, l'étude en amont de la cyclicité des ordonnancements d'avère être une nécessité.

L'autre aspect important dans ce contexte est l'étude de l'évolutivité de l'ordonnancement. Que ce soit dans le contexte monoprocesseur ou multiprocesseur, un aspect transverse de l'analyse est l'étude de l'évolutivité des applications. Les évolutions peuvent concerner l'ensemble des tâches, le noyau ou bien l'architecture cible. L'objectif est de proposer des solutions robustes et efficaces. Les problèmes que nous aborderons seront les suivants :

– Évolution de l'application : il s'agit d'adopter une approche modulaire de l'ordon-

nancement. Dans quelle mesure peut-on, lors de certaines opérations telles que l'adjonction ou la suppression d'une tâche ou bien la modification de certains paramètres de la tâche (la durée d'exécution ou de la date de réveil par exemple), réutiliser les résultats précédents pour dériver les nouvelles séquences,sans forcément tout recalculer.

- Évolution de l'architecture : il s'agit de prendre en compte les pannes matérielles. Comment concevoir l'application, comment la reconfigurer en cas de pannes de l'un des processeurs. Nous envisagerons différentes solutions et étudierons leur robustesse, en particulier pour les techniques d'ordonnancement global.
- Évolution du noyau : il s'agit d'envisager un fonctionnement multimodal permettant de passer d'un mode à un autre, i.e. d'une stratégie à une autre. Ceci peut au demeurant constituer une réponse à l'adjonction d'une tâche à l'application, ou bien à l'arrivée d'un flux de tâches apériodiques. Ceci permet également de proposer un mode de fonctionnement dégradé de l'application, par exemple en cas de surcharge ponctuelle.

La réponse à ces questions a fait l'objet de notre étude. Le manuscrit se compose de deux parties. La première partie constitue "l'État de l'art" et la seconde partie présente nontre contribution. Le plan de rédaction est plus détaillé dans le chapitre introductif.

NOTATIONS

$\mathbb{R}$ désigne l'ensemble des Réels

$\mathbb{N}$ désigne l'ensemble des entiers

- $\forall x \in \mathbb{R}, \lceil x \rceil$, est le plus petit entier supérieur ou égal à x ;
- $\forall x \in \mathbb{R}, \lfloor x \rfloor$, est le plus grand entier inférieur ou égal à x ;
- $\forall a, b \in \mathbb{N}$, avec $a < b$, $[a, b) = \{a, ..., b-1\}$ désigne l'ensemble des entiers naturels compris entre a et b, b exclu. Il s'en suit alors que $[a, b] = [a, b+1)$, $(a, b] = [a+1, b+1)$ et $(a, b) = [a+1, b)$.
- pour un ensemble A, $|A|$ désigne le cardinal de A.
- pour un ensemble A, $\wp(A) = \{B | B \subseteq A\}$ désigne l'ensemble des parties de A.
- pour un ensemble A, $\wp_m(A) = \{B | (B \subseteq A) \wedge (|B| \leq m)\}$ désigne l'ensemble des parties de A de cardinal inférieur ou égal à m.

INTRODUCTION GÉNÉRALE

0.1 Les systèmes temps-réel

Les systèmes informatiques temps-réel sont le plus souvent dédiés au contrôle de procédés. Ce sont des systèmes pour lesquels le temps est un critère de correction. En effet, la nature des procédés contrôlés implique un besoin d'un contrôle cohérent pour garantir leur sécurité. Il faut s'assurer que la vitesse du système informatique est adaptée à celle de l'évolution du procédé, donc que la durée entre le moment où une situation est repérée par les capteurs, et celui où les ordres en réaction sont émis par le système de contrôle est adaptée à la dynamique du procédé.

Dans la littérature, plusieurs définitions existent pour les systèmes temps-réel. Celle que nous retenons est la définition proposée par Gillies en 1995 qui dit qu'un système temps réel est un système dans lequel l'exactitude des applications ne dépend pas seulement de l'exactitude du résultat, mais aussi du temps auquel ce résultat est produit. Si les contraintes temporelles de l'application ne sont pas respectées, on parle de défaillance du système. Il est donc essentiel de pouvoir garantir le respect des contraintes temporelles du système.
L'élaboration d'une application temps réel passe donc par la donnée de spécifications à double visage : une spécification fonctionnelle comme pour toute application classique et une spécification temporelle. Par voie de conséquence, les applications temps-réel doivent être validées aussi bien fonctionnellement que temporellement.
A la différence donc des autres systèmes informatiques, l'aspect "temps" dans les systèmes temps-réel a une place importante. Il apparait sous la forme d'un "temps" de réaction ou d'une échéance à respecter. Les contraintes temporelles peuvent être de deux types :

- *souhaitées (voir la figure 2)* : on parle alors de *contraintes temporelles relatives* (système temps-réel "mou"). Les fautes temporelles sont tolérables (c'est le cas par exemple pour les jeux vidéo, les applications multimédia, la téléphonie mobile...),

mais leur multiplication et/ou des retards trop importants dégradent les performances de l'application ;

– *imposées (Voir la figure 1)* : on parle dans ce cas de *contraintes temporelles strictes* (système temps-réel "dur"). Les fautes temporelles ne sont pas tolérables (ceci concerne par exemple les applications du secteur de l'avionique, les véhicules spatiaux, les centrales nucléaires. . .)

Il existe des variantes de ces deux types de systèmes. La première variante est utilisée lorsqu'un résultat hors échéance voit sa qualité se dégrader jusqu'à un seuil minimum où il devient inacceptable. Ces types de systèmes temps-réel sont appelés des *systèmes temps-réel "fermes"* (firm real-time). Ce sont des systèmes temps-réel souples avec des manquements "occasionnels" d'échéances. La seconde variante est constituée de *systèmes temps-réel incrémental* dans lesquels le résultat gagne en qualité avec le temps (calcul scientifique). Les figures 3 illustrent des systèmes temps-réel "fermes" et les figures 4 des systèmes temps-réel incrémental.

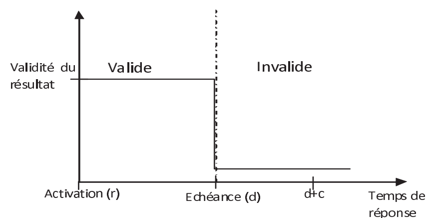


Figure 1 – Système temps-réel strict : *un résultat produit après l'échéance est invalide.*

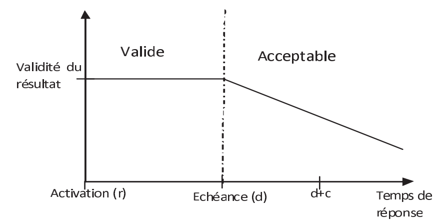


Figure 2 – Système temps-réel "mou" : *un résultat produit après l'échéance est accepté. Néanmoins le résultat produit est dégradé.*

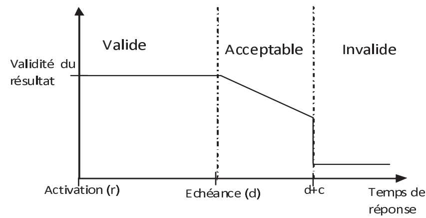


Figure 3 – Système temps-réel “ferme” : *un résultat dégradé est acceptable jusqu’à une certaine limite. Sur la figure, après $d+c$ le résultat produit n’est plus valide.*

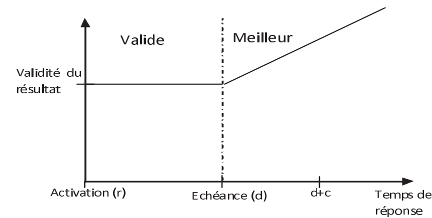


Figure 4 – Système temps-réel incrémental :
la qualité du résultat produit s’améliore avec le temps.

0.2 Le contexte d’exécution des systèmes temps-réel

L’automatisme au sein des systèmes temps-réel est généralement développé suivant deux types d’approches : l’approche *asynchrone* qui s’appuie généralement sur un exécutif multitâche temps-réel, et l’approche *synchrone* qui peut être abordée à l’aide d’un langage synchrone. Nous nous intéressons uniquement à l’approche asynchrone dans notre étude.
Dans ce contexte, les contraintes temporelles liées aux systèmes et la diversité des architectures conduisent à intégrer un exécutif temps réel dans les applications. L’exécutif apporte aux développeurs une interface standard quasiment indépendante du support matériel d’exécution et leur permet de concentrer leurs efforts sur les spécificités de leur application.

0.2.1 L’exécutif temps-réel

Un exécutif temps-réel (voir la figure 5) possède un noyau temps-réel complété de modules et de bibliothèques pour faciliter la conception de l’application temps réel. Ce sont : un module de gestion mémoire, de gestion des E/S, de gestion de timers, de gestion d’accès réseau et de gestion de fichiers.
Le noyau temps réel comporte les fonctionnalités de base nécessaires à l’exécution d’ap-

plications temps-réel : ordonnanceur, gestion de tâches, gestion des ressources, communications inter-tâches. On peut donc l'assimiler à un système d'exploitation plutôt limité mais performant. Sa caractéristique fondamentale est son déterminisme d'exécution avec des paramètres temporels fixés (temps de prise en compte d'une interruption, changement de contexte entre deux tâches, etc.).
Lors de la génération de l'exécutif, on choisit à la carte les bibliothèques en fonction des besoins de l'application temps-réel. Pour le développement, on a besoin d'une machine hôte et de son environnement de développement croisé (compilateur C croisé, utilitaires, debugger) ainsi que du système cible dans lequel on va télécharger (par liaison série ou par le réseau) l'application temps réel avec l'exécutif.
Il existe de nos jours beaucoup d'exécutifs temps-réel aussi bien dans l'industrie (VxWorks, pSos, VRTX, OSE, OS9, OSEK/VDX, LynxOS/LynuxOS, RTEMS, eCos) qu'au sein de la communauté scientifique (Mars [65], Spring [110], Hades [95], RTMach [113], DICK [27], variantes RTLinux [15, 51, 66, 115]). Des normes pour les interfaces de programmation système existent également (normes POSIX1003.1b et 1003.1c [55]). Tous ces exécutifs temps-réel ont en commun quelques fonctionnalités que nous présentons ci-dessous :

- *Interface de création des tâches* : certains exécutifs proposent des mécanismes pour associer la création de tâches à l'occurrence d'évènements de l'environnement (création orientée évènements, ou event-driven). D'autres ne permettent d'implanter que des tâches périodiques (exécution commandée par le temps, ou time-driven). Pour les applications où cela est permis, une tâche peut en créer une autre.
- *Gestion du temps* : le système peut proposer une fonctionnalité d'alarme différée, relativement au temps processeur occupé par la tâche (indépendamment des préemptions). Il peut aussi proposer une fonctionnalité de sommeil pendant une durée donnée, relativement à la date de début de la mise en sommeil (délai relatif qui accumule les erreurs dues à la résolution de l'horloge système), ou jusqu'à une date absolue (délai absolu). L'implantation des tâches périodiques par exemple dépend des méthodes de mise en sommeil disponibles.

– *Primitives de synchronisation* : les systèmes d'exploitation proposent en général des mécanismes de synchronisation basiques identiques, à savoir les sémaphores valués ou les verrous. Au delà, les systèmes peuvent adhérer à une norme, totalement ou partiellement, dans laquelle figurent une série de mécanismes de synchronisation supplémentaires (boite aux lettres, files de messages, verrous en lecture/écriture). Les systèmes peuvent également proposer leurs propres mécanismes de synchronisation, à la place ou en plus de ceux recommandés par les normes auxquelles ils peuvent décider de se conformer.
– *L'ordonnanceur* : pour définir la prochaine tâche à élire, le support d'exécution peut reposer sur un plan défini hors-ligne lors de la phase de conception, ou reposer sur un ordonnancement en-ligne s'appuyant sur des priorités relatives entre les tâches, ou en fonction de l'ordre d'activation (politique du premier arrivé, premier servi, ou FIFO, par exemple).
  Le système d'exploitation peut prendre ses décisions d'ordonnancement lors de chaque évènement système (blocage sur ressource, interruption matérielle), ou ne les prendre que lors d'une interruption d'horloge (tick scheduling).

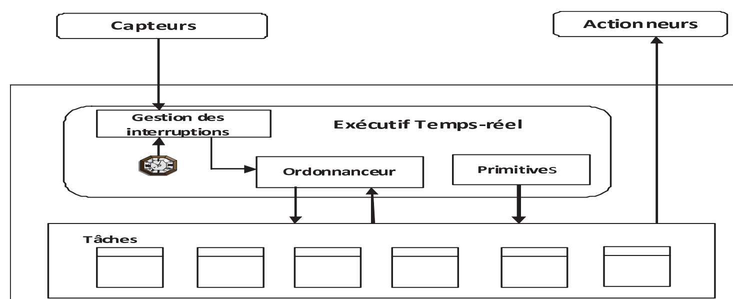


Figure 5 – Système temps-réel : exécutif et tâches temps-réel.

0.2.2 L'architecture matérielle

Les système temps-réel peuvent être caractérisés par rapport aux matériels avec lesquels ils interagissent. On distingue ainsi deux principales catégories de systèmes temps réel : les *systèmes temps-réel embarqués* et les *systèmes temps-réel organiques*. Les

systèmes temps-réel embarqués sont constitués d'applications temps-réel complètement encapsulées dans le matériel qu'elles pilotent ou contrôlent et en sont fortement dépendantes. Le matériel est souvent conçu et créé spécifiquement pour une application donnée. Dans ce matériel nous trouvons un ou plusieurs processeurs de type micro-contrôleur, des composants spécifiques de type ASIC (Application Specific Integrated Circuit), une alimentation électrique etc. Tout ce matériel est ensuite encapsulé dans un boîtier résistant à l'environnement de fonctionnement usuel.

Les systèmes temps-réel organiques sont peu dépendants du matériel sur lequel ils fonctionnent et peuvent en être détachés. Beaucoup d'applications de contrôle-commande s'exécutent par exemple sur des plateformes PC classiques. En revanche, il est toujours nécessaire de disposer de cartes d'entrées/sorties qui doivent souvent être synchronisées. Si l'architecture matérielle est composée d'un seul processeur on parle de *système temps-réel monoprocesseur*. Si elle est par contre composée de plusieurs processeurs partageant une horloge commune, on parle de *système temps-réel multiprocesseur*. Enfin dans de nombreux cas les applications peuvent être distribuées c'est à dire qu'elles s'exécutent sur plusieurs sites reliés par un ou des réseaux informatiques. On parle alors de *systèmes temps-réel distribués*.

0.3 Les applications temps-réel

Les systèmes temps-réel, le plus souvent embarqués, sont alors les composants de systèmes plus imposants dotés d'interactions dans le monde physique. C'est d'ailleurs là que réside la principale source de complexité des systèmes temps-réel. Le monde physique se comporte généralement de manière non déterministe : les évènements se produisent sans synchronisme, concurremment, dans un ordre imprévisible. Quelle que soit l'occurrence, le système temps-réel intégré doit réagir comme il convient et en temps voulu.

Le comportement concurrent du monde physique conduit par ailleurs à adopter une architecture multitâches pour ces applications car c'est la mieux adaptée pour répondre au comportement parallèle du procédé externe. Cette architecture logicielle multitâches

facilite la conception et la mise en œuvre et elle facilite l'évolutivité de l'application réalisée.
Généralement la mise en œuvre de la commande temps réel d'un procédé physique passe par les phases préalables suivantes :

1. à partir du cahier des charges de l'application, on recense tous les évènements et états qui traduisent l'évolution du comportement du procédé ;
2. on définit ensuite, sous forme d'algorithmes, les fonctions que doit remplir le système de contrôle en réponse à ces évènements et états pour assurer le suivi du procédé ;
3. les programmes de ces fonctions, appelées tâches temps-réel, sont alors implantés en mémoire du calculateur.

Les tâches temps-réel obtenues (voire une illustration sur la figure 6), qui constituent l'application ne sont pas des entités d'exécution indépendantes. Certaines tâches sont connectées vers l'extérieur pour les entrée/sorties, d'autres sont liées entre elles par des relations de synchronisation, de communication et de partage de ressources. Les tâches qui ne sont pas reliées entre elles sont dites *tâches indépendantes*.

Une tâche dont le réveil est conditionné par l'occurrence d'un signal périodique provenant d'une horloge est dite *périodique*. Dans ce cas, ses activations successives sont espacées d'une même durée appelée sa *période*. La période est déterminée en fonction de la précision du contrôle exigée par la dynamique du procédé à surveiller. Une application peut aussi mettre en oeuvre des tâches dont le réveil s'effectue de façon irrégulière, par exemple lors de l'occurrence d'un événement de nature sporadique. Si les activations des tâche surviennent à des moments aléatoires tels que la durée entre deux activations successives est bornée, ces tâches sont dite *sporadiques*. Mais si les activations surviennent à des moments totalement aléatoires, alors les tâches sont dites *apériodiques*. Ces concepts sont illustrés sur les figures 1.1, 1.2 du chapitre 1 où nous présentons un modèle d'activation d'une tâche périodique et d'une tâche sporadique.
Les dates de première activation des différentes tâches d'un système temps réel peuvent

être égales (nous considérons ici des systèmes multiprocesseurs, donc dotés d'une unique horloge commune à tous les processeurs), auquel cas le système de tâches est dit à *départs simultanés*. Dans le cas contraire il est dit à *départs différés*.

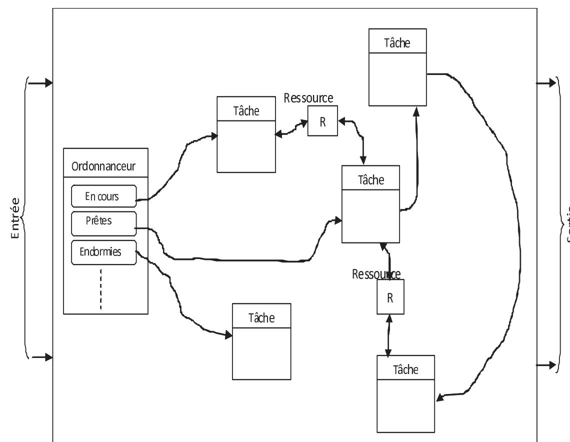


Figure 6 – Système de tâches :
les tâches peuvent partager des ressources critiques ou être reliées par des relations de précédence. L'ordonnanceur tient à jour une liste des tâches en exécution, des tâches prêtes et des tâches endormies. Dès qu'une tâche finit son exécution ou à l'arrivée d'un évènement, l'ordonnanceur est exécuté afin de choisir la prochaine tâche qui doit s'exécuter.

0.4 Le problème d'ordonnancement des applications temps-réel

Dans les systèmes temps-réel, les différentes tâches sont soumises à des contraintes temporelles. En effet, l'exécution de ces tâches doit respecter un délai maximum c'est à dire qu'aucune instance de tâche ne doit se terminer après son échéance. L'échéance d'une tâche est l'instant auquel l'exécution de la tâche doit se terminer. Le dépassement d'une échéance constitue une faute temporelle. Le but principal de l'ordonnancement est de permettre le respect des contraintes temporelles. Il consiste donc à définir une politique d'attribution du ou des processeurs de telle sorte qu'aucune faute temporelle ne soit commise.

Un ordonnancement est *valide*, si toutes les contraintes temporelles et de ressources sont respectées. Suivant les caractéristiques du modèle de tâches et du système, un ordonnancement valide peut exister ou non. Lorsqu'un tel ordonnancement existe, le système temps-réel est dit *ordonnançable*. Un système est dit ordonnançable par un algorithme donné si l'ordonnancement qui en résulte est valide. Pour le temps-réel strict, le critère de validité fondamental est que toutes les tâches temps-réel strict respectent toutes leurs contraintes temporelles en toute circonstance nominale de fonctionnement.
Valider une application temps-réel consiste à prouver que, quelle que soit l'évolution du procédé qu'elle pilote, elle ne se trouvera jamais en situation de violation de ses contraintes temporelles. L'algorithme de validation est désigné sous le nom de test d'ordonnançabilité dans la littérature [19]. Un test positif indique que pour tous les comportements possibles du système, les contraintes temporelles des tâches seront respectées.
Un ordonnancement est dit *optimal* s'il a une puissance d'ordonnancements maximale. Une stratégie d'ordonnancement est optimale dans une classe d'ordonnancement et pour une famille d'applications si elle est capable d'ordonnancer correctement n'importe quelle application (de la famille) pour laquelle il existe au moins un ordonnancement (de la classe) respectant toutes les contraintes temporelles.

Les *stratégies d'ordonnancement* temps-réel sont classées en deux grandes familles : les *stratégies hors-ligne* (avant exécution) et les *stratégies en-ligne* reposant sur un algorithme d'ordonnancement.
Utiliser une stratégie hors-ligne consiste à calculer au préalable une séquence valide qui décrit les événements d'activation des tâches et qui vérifie les contraintes des différentes tâches à ordonnancer. Le calcul de la séquence se fait en utilisant des techniques de simulation [60] ou des techniques de model-checking pour produire une séquence d'ordonnancement en accord avec les contraintes spécifiées [1, 2, 39]. La séquence ainsi calculée sera ensuite utilisée par un séquenceur durant la vie de l'application.
Utiliser une stratégie en-ligne consiste à exécuter un algorithme qui choisira à chaque instant la prochaine tâche à ordonnancer. Cet algorithme est implémenté dans l'ordon-

nanceur.
Dans ce contexte, la validation d'une application peut reposer soit, dans certains cas, sur des formules mathématiques, soit, si de telles formules n'ont pas pu être établies, sur des techniques de simulation. Ces techniques consistent à simuler le comportement du système sur la base des caractéristiques de l'ordonnanceur et du comportement pire cas de chacune des tâches. Mais cela ne conduit pas nécessairement à une simulation exacte du comportement du système car l'instabilité d'un ordonnancement, liée à la variation des paramètres temporels des tâches durant la vie du système, peut engendrer des situations qui n'auront jamais été simulées. En particulier, on travaille avec les pires durées des tâches, qui sont des majorants des durées effectives. Cela a conduit alors à l'utilisation de techniques d'*analyse pire cas*. Cette technique repose sur la caractérisation du pire comportement de l'application temps-réel. Trois techniques d'analyse pire cas existent dans la littérature : l'*analyse du facteur d'utilisation du processeur* [28, 81], l'*analyse de la demande processeur* [20, 104] et l'*analyse du temps de réponse* [70, 77]. Les deux dernières techniques sont souvent confondues sous le terme d'analyse de la demande de temps [83].

Pour rappel, l'ordonnancement consiste en l'attribution des tâches au(x) processeur(s). Contrairement aux ordonnancements monoprocesseur, en multiprocesseur, il faut ajouter une étape d'affectation des tâches aux processeurs et gérer éventuellement les migrations des tâches vers les autres processeurs. Cette affectation peut se faire de manière dynamique pendant l'exécution ou de manière statique avant l'exécution. Selon la stratégie adoptée pour l'affectation des tâches aux processeurs, on distingue principalement deux stratégies : la stratégie globale et la stratégie par partitionnement [52].
La stratégie d'ordonnancement est *partitionnée* si l'affectation des tâches aux processeurs est faite une fois pour toutes avant l'exécution et les décisions d'ordonnancement sont prises par un ordonnanceur local à chaque processeur (voire la figure 8). Dans ce modèle les tâches sont regroupées par processeur dès la phase de conception, et un algorithme d'ordonnancement local dérivé du cas monoprocesseur est employé sur chaque processeur. Les tâches sitôt affectées aux processeurs ne sont pas autorisées à migrer

d’un processeur à l’autre. Il s’agit donc d’une allocation statique.
L’ordonnancement est *global* si l’affectation se fait dynamiquement pendant l’exécution et les décisions d’ordonnancement sont prises par un seul ordonnanceur pour l’intégralité de la plateforme multiprocesseur (voir la figure 7). Il s’agit, dans cette stratégie, d’appliquer sur l’ensemble des processeurs la stratégie d’ordonnancement.

Leung et Whitehead ont montré dans [79] que dans la plupart des cas, ces deux approches sont incomparables en ce sens qu’il existe des systèmes ordonnançables selon l’une des méthodes mais pas par l’autre et réciproquement.

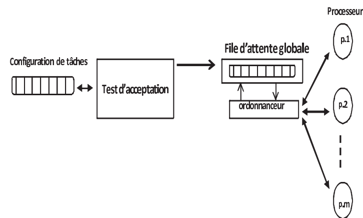


Figure 7 – Ordonnancement global

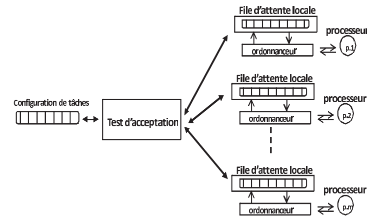


Figure 8 – Ordonnancement partitionné

Nous adoptons dans la suite du document l’approche globale qui semble plus adaptée à la gestion de l’évolutivité de l’application, qui nécessite une vision globale du système.

0.4.1 Les ordonnancements en-ligne

L’ordonnancement à l’exécution ou ordonnancement en-ligne consiste à implanter dans l’exécutif un ordonnanceur qui s’appuie sur un algorithme mettant en œuvre une politique spécifique. Chaque évènement (signal ou commutation) voit l’ordonnanceur élire la prochaine tâche en fonction de la politique choisie. La majorité des ordonnanceurs temps-réel en-ligne reposent sur la notion de priorité : lors de chaque décision d’ordonnancement, la tâche prête de plus haute priorité est élue. Les priorités peuvent être attribuées hors-ligne, auquel cas l’algorithme d’ordonnancement est dit à *priorités statiques* ou fixe [80, 81] ; ou en-ligne, auquel cas il est dit à *priorités dynamiques* [30, 31, 49, 81, 103].

Dans le cas des ordonnanceurs à priorités dynamiques, l'algorithme doit déterminer les priorités relatives des différentes tâches prêtes. Ce type d'ordonnanceur est donc exigeant en terme de services que doit fournir l'exécutif. Celui-ci doit entre autre pouvoir gérer des files de tâches à ordonnancer, et doit pouvoir déterminer quand une décision d'ordonnancement est nécessaire. Il en résulte par là même des appels fréquents à l'ordonnanceur ce qui accroit le surcout d'exécution engendré.
Dans un contexte d'exécution sur plusieurs processeurs, il n'existe pas d'algorithme en ligne optimal dans le cas général, même pour les tâches indépendantes. Cependant, si on considère le modèle classique des tâches à départs simultanés et échéances sur requêtes (l'échéance d'une instance correspond à la date d'activation de l'instance suivante, la contrainte consistant à éviter la réentrance), il existe une famille d'algorithmes optimaux, à laquelle nous allons nous intéresser, la famille des ordonnancements P-équitables.
Pour la validation temporelle des application dans ce contexte, dans les cas favorables, on a des critères analytiques, mais sinon, il faut passer par la simulation.

En pratique, les exécutifs les plus courants et les standards disponibles les plus utilisés, fournissent les fonctionnalités nécessaires à l'ordonnancement à priorité fixe (Le profile de Ravenscar [11, 24]).
Une présentation des algorithmes d'ordonnancement en ligne classique peut être trouvée dans [32] pour le lecteur intéressé.

0.4.2 Les ordonnancements hors-ligne

Nous avons vu que les stratégies en-ligne ne sont pas optimales en général et on ne dispose pas de critères d'ordonnançabilité dans tous les cas. De plus, il y a des problèmes quand on se tourne vers la simulation car simuler avec les pires temps de réponse ne donne pas nécessairement des résultats concluants. Il faut donc envisager d'autres techniques d'ordonnancement. Les techniques hors-ligne offrent une autre approche.
Les ordonnancements hors-ligne reposent sur la définition d'un plan statique. Les méthodes pour l'établissement du plan sont basées sur l'énumération exhaustive des ordonnancements possibles de la configuration de tâches donnée. L'énumération permet,

dès lors que le configuration est ordonnançable, d'exhiber une séquence valide qui sera implantée dans un séquenceur. Les stratégies d'ordonnancement hors-ligne possèdent l'avantage de se contenter d'un support d'exécution très réduit (l'accès à une base de temps suffit). De plus, l'environnement n'influence pas le déroulement du plan, ce qui rend le système robuste en ce sens que les fluctuations de durées effectives d'exécution ne mettront pas l'application en péril. Enfin, la garantie que le système respectera toutes ses contraintes temporelles est une propriété facile à appréhender : il suffit que le plan soit vérifié hors-ligne, et que les évaluations des temps d'exécution soient correctes. Ceci fait de cette solution une solution rigoureuse stricte.

0.5 Les problèmes envisagés

Nous souhaitons à terme pouvoir proposer des techniques d'ordonnancement capables de gérer l'évolutivité des systèmes. Cela nous a conduit à considérer un certain nombre de problèmes dont l'étude a constitué le cœur de cette thèse.

0.5.1 Le problème de la cyclicité

Pour réaliser des études basées sur des simulations permettant d'illustrer nos propositions, et d'en évaluer la pertinence, nous devons déterminer la durée sur laquelle les simulations doivent être réalisées. Par ailleurs, bien que les stratégies d'ordonnancement hors-ligne soit optimales, elles se heurtent néanmoins à un problème complexe : la connaissance de la taille de la séquence à construire est nécessaire. En effet, il faut être capable de décrire de manière finie un comportement infini i.e un comportement en régime permanent. Cela implique de répondre en amont aux deux questions fondamentales suivantes :

- quelle est la taille de la séquence à fournir au séquenceur ?
- comment engendrer le régime permanent à partir de cette séquence ?

Il est donc nécessaire de se ramener à des plans périodiques ou à des plans qui supposent connues toutes les dates d'arrivées (éventuellement non périodiques) de toutes les tâches.

La périodicité des tâches permet de limiter la taille de la séquence d'ordonnancement à produire car l'ordonnancement lui même devient périodique à partir d'une certaine date. Il a été établi dans les travaux de [42, 78] que la période de l'ordonnancement est égale au *ppcm* des périodes des tâches encore appelé *hyperpériode*. Le problème qui se pose est de déterminer quand commence la phase périodique de l'ordonnancement. Ce problème est connu sous le nom de *problème de la cyclicité*.
Une solution au problème de cyclicité permettrait alors de limiter l'espace d'exploration si l'approche hors-ligne basée sur un modèle est utilisée. La profondeur de cet espace d'exploration est encore appelée *intervalle d'étude*.
Si l'approche en-ligne est utilisée, il est souvent nécessaire de connaitre la taille de cet intervalle d'étude pour l'analyse d'ordonnançabilité ou pour pouvoir réaliser des simulations sur une durée pertinente.

Le problème de la cyclicité a été résolu dans le contexte monoprocesseur dans les travaux de [37, 78]. Les travaux de [78] donnent une borne de la taille de la séquence à produire pour des systèmes de tâches indépendantes ordonnancées par ED et [37] améliore cette borne et la généralise à des systèmes quelconques, pour n'importe quelle stratégie.
Si la phase cyclique encore appelée *régime permanent* de la séquence démarre à l'instant $t_c + 1$, alors la séquence à produire doit correspondre à la fenêtre temporelle $[0, t_c + T + 1]$ qui constitue l'*intervalle d'étude* du système (où T est l'hyperpériode du système). Pour un système de tâches à départs simultanés, toutes les tâches seront réactivées à l'instant T. L'intervalle d'étude dans ce cas est donc $[0, T]$.
La connaissance de l'instant de début de la phase périodique devient plus difficile quand l'application est composée de tâches à départs différés. Dans le contexte des ordonnancements monoprocesseur, les travaux de [37] ont donné une borne à l'entrée dans la phase cyclique pour un système de tâches donné. Cette borne dépend des caractéristiques temporelles du système de tâches et est indépendante de la stratégie choisie.

Dans le cas des systèmes multiprocesseurs, le problème n'a été résolu que de façon limitée, pour des systèmes de tâches indépendantes et les stratégies à priorités fixes [33, 38, 42].
Très peu de résultats existent concernant la problématique de la cyclicité dans ce contexte. Le régime permanent commence à l'instant 0 si l'application consiste en un système de tâches à départs simultanés. Par contre pour les systèmes de tâches à départs différés, aucune étude n'a donné la date de début du régime permanent de manière générale.
Dans notre approche pour l'étude de la cyclicité en multiprocesseur, nous nous intéresserons aux tâches indépendantes et à des ordonnancements appartenant à une classe que nous introduirons, la classe des *ordonnancements monotones*. Elle contient toutes les stratégies classiques : les priorités fixes, EDF, LLF, et les stratégies P-équitables. Notre objectif est de caractériser d'un point de vue algorithmique l'instant d'entrée dans le cycle, en observant les moments d'inactivité des processeurs.

0.5.2 La gestion des pannes

Un système temps-réel est une association logiciel/matériel où le logiciel permet, entre autre, une gestion adéquate des ressources matérielles en vue de remplir certaines tâches ou fonctions dans des limites temporelles bien précises. La nature critique des tâches ou fonctions fait que l'une des préoccupations majeure lors du développement et de l'exploitation de tels systèmes est d'assurer une certaine robustesse face aux pannes possibles du support matériel.
Les termes *faute*, *erreur*, et *défaillance* sont souvent utilisés comme synonymes. Toutefois, les définitions suivantes [76] sont plus appropriées dans le contexte des problèmes que nous étudions. Une *faute* est un défaut dans un composant (panne) ou une tâche du système. Une faute se manifeste alors comme un état erroné du système. Quand cet état erroné interne inattendu a des répercutions sur le comportement extérieur du système, il est appelé une *erreur*. Une *défaillance* du système se produit lorsque le service rendu s'écarte de la spécification du système. Les lecteurs intéressés peuvent se référer à [26, 76] pour une classification des fautes.
Dans notre étude nous nous intéressons au problème de la panne d'un processeur. La

question à laquelle nous voulons répondre est : comment garantir qu'un système de tâches ordonnançable le reste même si une panne processeur survient en cours de fonctionnement ? L'approche que nous utilisons est la *redondance temporelle*. Cela consiste en la re-exécution de la tâche affectée par la panne processeur. Le re-exécution peut être totale ou partielle. La re-exécution est totale quand l'instance de la tâche est reprise entièrement. La re-exécution est partielle quand l'instance est reprise à partir d'un état qui a été préalablement sauvegardé. Cela suppose qu'une politique de sauvegarde/reprise ait été mise en oeuvre. Les lecteurs intéressés par les techniques de reprises pourront se référer aux travaux de thèse de Punnekat [96]. Une autre approche consiste à utiliser la redondance spatiale : chaque tâche est dupliquée et les deux versions s'exécutent en parallèle sur des processeurs distincts. Cette approche nécessite donc de doubler le nombre de processeurs. Nous avons choisi une approche permettant de limiter les surcoûts.
Nous avons considéré les mécanismes de reprise après la panne d'un processeur. Nous avons considéré dans cette étude le cas où un seul processeur tombe en panne mais nos résultats peuvent se généraliser au cas où plusieurs pannes surviennent. Nous avons envisagé différentes stratégies de reprise totale ou partielle, pour les stratégies d'ordonnancement EDF et PF. Pour les stratégies équitables, nous avons envisagé la reprise totale avec modification des paramètres temporels. Pour cela, nous avons donc été amenés à étudier plus précisément le fonctionnement de ces stratégies.

0.5.3 Etude des ordonnancements P-équitables dans un contexte plus large

Les algorithmes d'ordonnancement P-équitables sont des algorithmes à priorités dynamiques optimaux dans le contexte des ordonnancements multiprocesseur pour des tâches indépendantes, à départs simultanés et à échéances sur requêtes.
Pour prendre en compte la tolérance à une panne processeur, des ré-aménagements de ce modèle d'ordonnancement s'avèrent nécessaires car le modèle de tâches considéré n'est plus à départs simultanés et à échéances sur requêtes. En effet, une panne peut conduire à la re-exécution d'une ou plusieurs tâches, ce qui entraine un décalage de ces tâches et cela revient à gérer un système à départs différés. De plus, pour permettre la reprise des tâches, il faut considérer des échéances contraintes. Dans la littérature, des travaux

supplémentaires ont proposé des extensions au contexte d'utilisation des ordonnancements P-équitables. Les cadres envisagés ne correspondent cependant pas exactement à notre approche. Nous avons donc redéfini la notion d'ordonnancement P-équitable dans un contexte d'ordonnancement de tâches à départs différés et à échéances contraintes puis nous avons proposé une condition suffisante d'ordonnançabilité. Nous avons enfin procédé à des simulations pour étudier la pertinence de notre condition.

0.5.4 Adjonction de nouvelles tâches

La panne d'un processeur, est un évènement qui peut survenir dans la vie d'un système multiprocesseur. Il est donc nécessaire de disposer de méthodes performantes d'analyse d'impact sur l'ensemble du système pour la maitrise du surcout engendré par la politique de tolérance à la panne processeur. De plus les systèmes temps-réel se caractérisent par une interaction continue avec leur environnement et doivent satisfaire non seulement des exigences fonctionnelles prévues a priori mais également celles non prévues.
Il s'agit donc d'étendre la capacité du système pour s'adapter à un nouvel environnement. Une solution consiste en la reprise de la tâche impactée par la panne processeur en réponse à un problème lié à une panne processeur. Cette tâche peut être reprise comme une nouvelle tâche. L'étude d'une telle solution peut se généraliser en l'étude de l'adjonction de nouvelles tâches dans le système ou la suppression d'une tâche du système. L'étude d'un fonctionnement multimodal du système permettant l'évolution des exigences fonctionnelles au cours de la vie de l'application, peut également être intéressante.
Nous étudierons des protocoles d'adjonction de tâches pendant la phase d'exploitation du système. Nous nous intéresserons tout d'abord à l'adjonction d'une tâche dans un système de tâches ordonnancé selon une politique d'ordonnancement P-équitable. Notre intérêt se portera plus particulièrement dans ce cas à la gestion des flux d'apériodiques. Nous proposerons un test d'acceptation de complexité polynomiale.
L'algorithme d'acceptation que nous proposons repose sur le principe de répartition régulière des temps creux processeur. Afin de pouvoir traiter des systèmes de tâches in-

teragissantes, nous nous tournons vers une stratégie hors-ligne. Nous considérons une approche à base de modèle s'appuyant sur les réseaux de Petri. Nous avons adapté l'approche proposée dans [60] au contexte multiprocesseur, puis nous avons proposé des techniques de construction de séquences de manière à intégrer en particulier le critère de répartition régulière des temps creux. Ceci nous permet alors d'étendre notre routine d'acceptation à ce contexte.

0.6 Organisation du document

Le document est structuré en deux grandes parties, la première présente le contexte général de nos travaux et l'état de l'art. Elle se compose de deux chapitres. Le premier traite de l'ordonnancement multiprocesseur. On y introduit toutes les notions générales, ainsi que les algorithmes P-équitables. Le second chapitre présente l'approche hors-ligne à base de réseaux de Petri, tirée de [60]. La seconde partie présente le travail réalisé. Elle est organisée en 5 chapitres. Le premier traite du problème de la cyclicité. Le second s'intéresse au problème de la reconfiguration en cas de panne d'un processeur. Les deux chapitres suivants mettent en place des outils utiles pour résoudre les problèmes de reprise : le chapitre 5 s'intéresse aux ordonnancements P-équitables, le chapitre 6 présente notre routine d'acceptation de tâches apériodiques, d'une part dans le cas de systèmes de tâches indépendantes, et d'autre part pour des systèmes de tâches interagissantes. Dans le premier cas, les tâches périodiques sont ordonnancées par un algorithme équitable, dans le second cas, nous présentons une stratégie mixte : les tâches périodiques sont ordonnancées hors-ligne, à l'aide d'une approche à base de réseau de Petri, et les tâches apériodiques sont acceptées en ligne. Enfin, nous donnons quelques résultats concernant l'adjonction d'une tâche périodiques au système ainsi que le fonctionnement multimodal de l'application dans le chapitre 7 où nous concluons par ailleurs.

Première partie

Etat de l'art

Dans cette partie, nous présentons de manière générale les systèmes temps-réel qui sont des systèmes pour lesquels l'exactitude des applications ne dépend pas seulement de l'exactitude des résultats, mais aussi du temps auquel ce résultat est produit. Nous nous intéressons plus particulièrement à l'ordonnancement de ces applications. Dans le premier chapitre de cette partie, nous présentons les différentes politiques d'ordonnancement des applications temps-réel sur des plateformes multiprocesseurs. Nous présentons les limites de ces politiques d'ordonnancement quand on passe d'une plateforme monoprocesseur à une plateforme multiprocesseur. Nous terminons la partie par un second chapitre où nous présentons plus particulièrement les politiques d'ordonnancement hors-ligne à base de modèle qui proposent une alternative aux stratégies d'ordonnancement en ligne.

CHAPITRE 1

ORDONNANCEMENT MULTIPROCESSEUR

1.1 Contexte d'exécution

1.1.1 Contexte matériel

L'évolution de la puissance des processeurs est insuffisante de nos jours au regard des besoins des applications temps réels. Par conséquent les applications temps-réel complexes sont déployées sur des architectures multiprocesseurs, afin de leur permettre de satisfaire leurs contraintes temporelles. Un cas particulier des plate-formes multiprocesseur nous intéresse dans cette thèse : les plate-formes composées de plusieurs *processeurs identiques dotés d'une horloge commune*. Sur ces plate-formes, les processeurs ont la même puissance de calcul. Il existe des architectures matérielles composées de plusieurs processeurs, chacun des processeurs étant caractérisé par sa propre capacité de calcul. Lorsqu'un traitement met t unités de temps à s'exécuter sur un processeur, ce même traitement demandera $s \times t$ unités de temps pour s'exécuter sur un autre processeur de la plateforme, le facteur s étant constant et connu. On parle alors de *processeurs uniformes*. Les processeurs identiques sont donc un cas particulier des processeurs uniformes qui vérifient $s = 1$ pour tout couple de processeurs.

Dans la suite nous considérons que chaque processeur possède sa propre mémoire privée et peut accéder en temps constant à une mémoire commune. Sous cette hypothèse, le coût de communication inter processeurs est généralement négligé, contrairement au cas réparti [72].

L'étude des problèmes d'ordonnancement doit donc être adaptée à ce type d'architecture. Nous adoptons alors les hypothèses classiques suivantes en plus :

- un processeur ne peut exécuter qu'une seule tâche à la fois ;
- une tâche ne peut pas être exécutée par plusieurs processeurs au même instant. Ceci signifie que les étapes de parallélisation ont été faites préalablement et donc les tâches que nous considérons sont supposées non parallélisables

1.1.2 Le modèle de tâche

Généralement, une application temps-réel est une application multitâches. Elle est modélisée par un ensemble de $n \in \mathbb{N}$ tâches $\tau = \{\tau_1, \tau_2, ...\tau_n\}$ appelé *configuration de tâches* ou *système de tâches*. La nature des tâches découle directement de l'objectif de l'application c'est-à-dire de son cahier de charge. Si les activations de tâches sont périodiques, alors les tâches sont dites *périodiques*. Par contre si les activations des tâches sont aléatoires, alors elles sont dites *apériodiques*. Il peut arriver que pour une tâche donnée, deux activations successives soient espacée d'une durée bornée, dans ce cas la tâche est dite *sporadique*. Les figures 1.1 et 1.2 illustrent respectivement les activations d'une tâche périodique τ_i et celles d'une tâche sporadique.

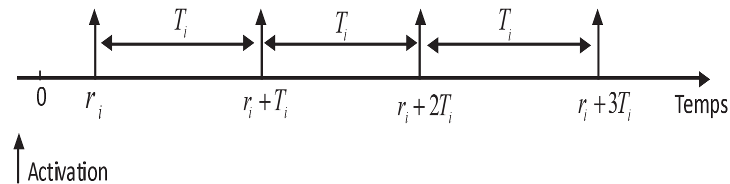


Figure 1.1 – Activations périodiques de période T_i

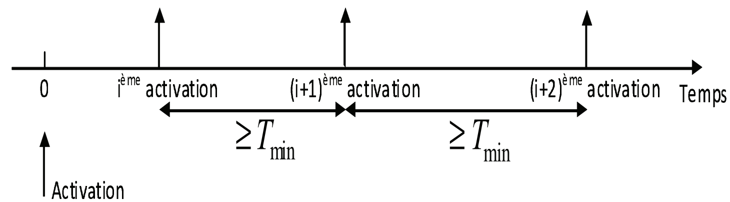


Figure 1.2 – Activations sporadiques

Une tâche de l'application est appelée tâche temps-réel. Une tâche temps-réel périodique τ_i est caractérisée par quatre paramètres temporels qui sont (r_i, C_i, D_i, T_i). r_i est la date de première activation de la tâche, C_i est sa pire durée d'exécution (durée de traitement sans interruption sur un processeur), D_i son délai critique et T_i sa période.

Une tâche sporadique est quant à elle caractérisée par 3 paramètres temporels (C_i, D_i, T_i) : sa date de première activation n'est pas connue. Enfin, une tâche apériodique est caractérisée par deux paramètres temporels (C_i, D_i) : elle s'exécute une seule fois à un instant non connu a priori.

L'*instance d'une tâche* correspond à son exécution sur une *période d'activité* (une période d'activité correspond à l'intervalle de temps séparant deux activations succéssives). Si une tâche périodique τ_i a un délai critique égal à sa période c'est à dire $D_i = T_i$, elle est dite à *échéances sur requêtes*. Si le délai critique est inférieur à la période, c'est-à-dire $D_i < T_i$, la tâche est dite à *échéances contraintes*. Lorsque D_i et T_i sont indépendantes, la tâche est dite à *échéances arbitraires*.

On appelle *facteur d'utilisation* du processeur pour la tâche τ_i le taux d'activité du processeur pour l'exécution des instances successives de la tâche. Le facteur d'utilisation du processeur de la tâche τ_i est noté U_i et est donné par la relation $U_i = \frac{C_i}{T_i}$. Le facteur d'utilisation du processeur de l'application est le taux d'activité du processeur pour l'exécution du flot d'instances des tâches de l'application. Il est noté U et est donné par la relation $U = \sum_{i=1}^{i=n} \frac{C_i}{T_i}$.

L'utilisation réelle du processeur par une tâche à échéances contraintes se situe entre son activation et son échéance. On introduit alors le *facteur de charge* d'une tâche τ_i noté CH_i qui est défini par $CH_i = \frac{C_i}{D_i}$. Si la tâche est à échéances sur requêtes, le facteur de charge et le facteur d'utilisation coïncident. Le facteur de charge de l'application est la grandeur $CH = \sum_{\tau_i \in \tau} \frac{C_i}{D_i}$.

Pour une configuration de n tâches $\tau = \{\tau_1, \tau_2, ... \tau_n\}$, on appelle *hyperpériode* T le *ppcm* des périodes des tâches c'est-à-dire $T = ppcm(T_1, T_2, ..., T_n)$ et $r = max\{r_i\}_{\tau_i \in \tau}$ la plus grande date de première activation.

1.2 Ordonnancement global versus partitionné

Comme nous l'avons précisé dans le chapitre d'introduction, dans un contexte d'ordonnancement multiprocesseur, il faut ajouter une étape d'affectation des tâches aux processeurs et gérer éventuellement les migrations des tâches vers les autres proces-

seurs. Cette affectation peut se faire de manière dynamique pendant l'exécution auquel cas on parle d'ordonnancement global ou être faite a priori et de façon définitive avant l'exécution, et dans cas on parle d'ordonnancement partitionné.

1.2.1 Ordonnancement global

Il s'agit, dans cette stratégie, d'appliquer sur l'ensemble des processeurs la stratégie d'ordonnancement globalement. A tout instant, les tâches prêtes à être exécutées et de plus haute priorité sont affectées aux processeurs jusqu'à concurrence du nombre de processeurs. Si toutefois il y a moins de tâches actives que de processeurs, certains processeurs seront laissés oisifs. Dans cette stratégie la migration des tâches est possible. La migration est forte ou totale lorsqu'une tâche peut commencer son exécution sur un processeur et la terminer sur un autre processeur. La migration est faible si une tâche doit terminer son exécution sur le processeur sur lequel elle a commencé. Cependant deux instances différentes peuvent être exécutées sur deux processeurs différents. Nous nous intéressons dans notre travail au cas où la migration est totale.

L'ordonnancement des applications temps réel sur des plateformes multiprocesseurs suivant une stratégie globale pose de nombreux problèmes. Dans un contexte d'ordonnancement en-ligne, les travaux de [48, 67] ont montré qu'il ne peut pas exister d'algorithme d'ordonnancement optimal dans le cas général. De plus, les travaux de [79] ont montré que le problème de l'ordonnancement sur une architecture multiprocesseur suivant une stratégie globale est NP- complet (voir [109] pour un panorama complet de la complexité de ce problème) et par ailleurs, les travaux de [82, 109] ont montré que dans ce contexte on peut se heurter à des anomalies d'ordonnancement même lorsque l'on ne considère que des tâches indépendantes. Par exemple :

- Le fait de considérer exactement connus les paramètres temporels d'une tâche (durée d'exécution pire-cas, période, etc.) peut conduire à des erreurs d'analyse [8, 35] :
  - l'exemple suivant que nous avons pris dans la thèse de David Decotigny [46] montre que la terminaison plus tôt que prévue d'une tâche peut conduire à une

anomalie d'ordonnancement illustrée dans les figures 1.3 et 1.4. Ceci montre que valider par simulation avec la pire durée ne suffit pas. Il faut figer la séquence dans ce cas, donc passer dans un mécanisme hors-ligne, i.e le calcul d'une séquence et non pas d'utilisation d'un algorithme.

- dans un contexte d'ordonnancement à priorités fixes, une période plus grande que prévue sans modification de la priorité peut conduire à une faute temporelle. Cela peut s'observer sur les figures 1.5 et 1.6. Sur la figure 1.5 la séquence produite est valide sur deux processeurs pour le système de tâches. En modifiant la période de la tâche τ_1 qu'on ramène à 5, la séquence produite n'est plus valide comme l'illustre la figure 1.6. Cela est dû au fait que l'ordre d'exécution de la tâche τ_2 ne peut pas changer. Il aurait fallu réaménager les priorité des tâches pour obtenir une séquence valide.

– La diminution de la durée d'exécution (implicitement du facteur d'utilisation) pour un système non ordonnançable ne garantit pas toujours qu'il deviendra ordonnançable. Cela s'appelle l'effet Dhall introduit par Dhall et Liu dans [50]. L'exemple suivant illustre bien ce phénomène : soit un système τ de $m+1$ tâches périodiques ordonnançables sur une plateforme de m processeurs par *RM*. Les tâche de τ sont telles que : $\forall \tau_i \in \tau$ avec $i = 1 \dots m$, $T_i = 1$ et $C_i = 2\varepsilon$. La tâche τ_{m+1} est telle que $T_{m+1} = 1+\varepsilon$ et $C_{m+1} = 1$. Toutes les tâches s'activent à $t = 0$. Les tâches $\tau_i,\ i = 1 \dots m$, prendront les m processeurs car elles sont les plus prioritaires et s'exécuteront pendant 2ε unités de temps. Ensuite la tâche τ_{m+1} pourra prendre un processeur à l'instant 2ε. Comme sa durée d'exécution est 1, elle terminera à l'instant $1+2\varepsilon$ alors que son échéance est $1+\varepsilon$. Donc la tâche τ_{m+1} ratera son échéance. Nous avons $U = 2\varepsilon m + \frac{1}{1+\varepsilon}$. En maintenant m constant et en faisant décroitre ε, on obtient un système de tâche dont le facteur d'utilisation tend vers 1 mais non ordonnançable. Ce phénomène s'observe même si on considère un ordonnancement EDF [50].
– Les travaux de [8, 75] montrent que la notion d'*instant critique* pour une tâche introduit dans le cas monoprocesseur par [81] n'est plus valable quand on passe en multiprocesseur(l'instant critique pour une tâche τ_i correspond à l'instant auquel

elle est activée simultanément avec l'ensemble des tâches plus prioritaires qu'elle. Le scénario ainsi décrit correspond, dans le cas monoprocesseur au scénario pire cas pour la tâche τ_i et le temps de réponse pour cette instance correspond au temps de réponse le plus long encore appelé temps de réponse pire cas). Dans le cas multiprocesseur, ceci ne correspond pas nécessairement à la pire durée, comme l'illustre la figure 1.7. D'après les travaux de [81], l'instant critique de la tâche τ_3 est 0 et alors sa première instance devrait avoir le plus long temps de réponse. Sur la figure, on remarque que le temps de réponse de la première instance de la tâche τ_3 est 3 alors que le temps de réponse de la seconde instance est 4. Donc cette définition du pire cas ne tient plus en multiprocesseur et cette situation a pour conséquence l'absence d'analyse pire cas en multiprocesseur.

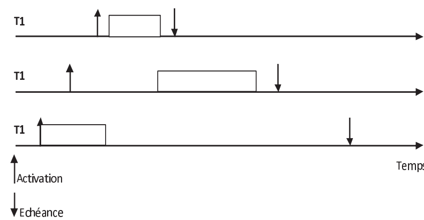


Figure 1.3 – Ordonnancement valide

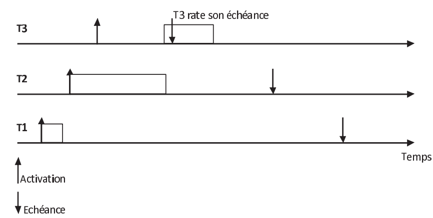
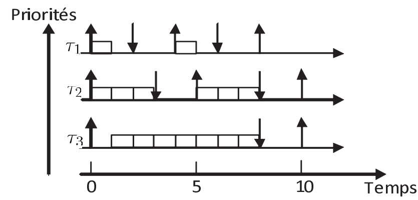


Figure 1.4 – Ordonnancement non valide :

la diminution du temps d'exécution de la tâche T_1 a entraîné le dépassement d'échéance de la tâche T_3 (figure de droite) alors que l'ordonnancement était valide(figure de gauche). Dans cet exemple l'ordonnancement est non préemptif et au plus tôt sur un processeur et la priorité est donnée à la tâche ayant l'échéance la plus courte. Notons que l'exemple ne correspond pas à notre contexte. Il illustre juste que dans le cas le plus simple, considérer comme connus les paramètres d'une tâche peut conduire à des anomalies d'ordonnancement.



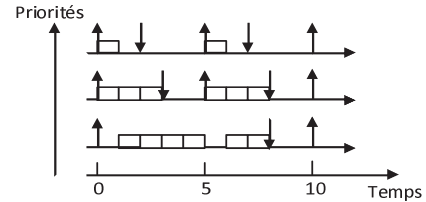


Figure 1.5 – Anomalie d'ordonnancement :

Figure 1.6 – Anomalie d'ordonnancement :

sur la figure de gauche, la séquence d'ordonnancement du système de tâches $\tau = \{\tau_1(0,1,2,4), \tau_2(0,3,3,5), \tau_3(0,7,8,10)\}$ *est valide. Sur la figure de droite, le système de tâches devient* $\tau = \{\tau_1(0,1,2,5), \tau_2(0,3,3,5), \tau_3(0,7,8,10)\}$ *après le changement de la période de* τ_1. τ_3 *rate son échéance, l'ordonnancement n'est plus valide .*

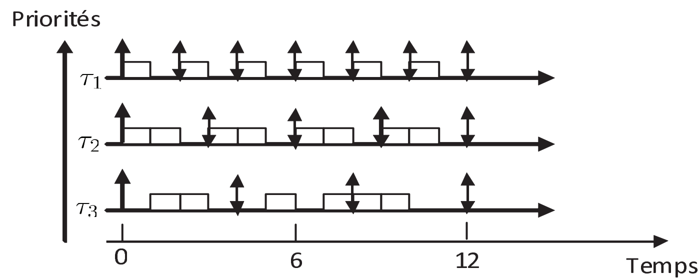


Figure 1.7 – Anomalie d'ordonnancement :

Sur cette figure, la séquence d'ordonnancement du système de tâches $\tau = \{\tau_1(0,1,2,2), \tau_2(0,2,3,3), \tau_3(0,2,4,4)\}$ *montre que la notion d'instant critique défini dans les travaux de [81] n'est plus valide quand on passe sur du multiprocesseur. En effet, le temps de réponse de la tâche* τ_1 *sur sa première instance est 3 alors que l'instant* $t = 0$ *correspond à son instant critique. Sur sa deuxième instance par contre, le temps de réponse de la tâche est 4.*

Malgré ces difficultés (complexité et anomalies), lorsque l'on considère les plateformes multiprocesseurs identiques, des travaux se sont intéressés à un aménagement des algorithmes d'ordonnancement *RM* [81] et *EDF* [81]. Les travaux de [9] proposent une condition suffisante pour l'ordonnançabilité avec *RM*. Les travaux de [107] ont montré

que sur une plateforme de m processeurs identiques et pour un système de tâches indépendantes, si le facteur d'utilisation de chaque tâche vaut au plus $\frac{m}{2m-1}$ et le facteur d'utilisation totale est inférieur ou égal à $\frac{m^2}{2m-1}$, alors ce système de tâches est ordonnançable par EDF. Ils ont par la suite proposé l'algorithme $EDF - US[m/(2m-1)]$ qui est un aménagement de l'algorithme EDF et $RM - US[m/(2m-1)]$ qui est un aménagement de RM, et ont montré que ces algorithmes ordonnancent respectivement tout système de tâches indépendantes dont le facteur d'utilisation total est inférieur ou égal à $\frac{m^2}{2m-1}$ et $\frac{m^2}{3m-2}$. Ces résultats ont été généralisés dans [12] pour l'algorithme EDF, qui montre que EDF ordonnance tout système de tâches indépendantes dont le facteur d'utilisation total U est tel que $U \leq m(1-u_{max}) + u_{max}$ où $u_{max} = max\{U_i | \tau_i \in \tau\}$. Baker, dans [13] donne une condition générale d'ordonnançabilité avec EDF sur une plateforme de m processeurs identiques suivant la stratégie globale. Dans ses travaux, il montre qu'un système de n tâches périodiques indépendantes $\tau_1, \ldots \tau_n$ est ordonnançable sur m processeur par EDF si $\sum_{i=1}^{i=n} \frac{C_i}{T_i}(1 + \frac{max\{0, T_i - D_i\}}{D_{min}}) \leq m - \hat{\lambda}(m-1)$, où $\hat{\lambda} = max\left\{\frac{C_i}{min\{D_i, T_i\}}, i = 1, \ldots, n\right\}$ et $D_{min} = min\{D_i, i = 1 \ldots n\}$.
L'ordonnançabilité par EDF dépend ici de $\hat{\lambda}$. Plus la valeur de $\hat{\lambda}$ décroît, plus le facteur d'utilisation total croît. Ce résultat a été étendu aux algorithmes d'ordonnancement RM dans [14]. Dans tous les cas, pour que le système soit ordonnançable, il faut surdimensionner le système entraînant ainsi une perte de temps processeur.

Baruah et al. dans [18] ont étendu les algorithmes d'ordonnancement fluides au cas multiprocesseur identiques pour les tâches périodiques indépendantes. Ils ont montré l'existence d'une famille d'algorithmes de complexité polynomiale si on considère des tâches à départs simultanés à échéances sur requêtes. Ces algorithmes sont appelés *algorithmes d'ordonnancement P-équitables* (*PFair*). Les fondements théoriques de cette famille d'algorithmes sont établis dans [16]. Nous verrons plus en détail ces algorithmes en section 1.4 et dans le chapitre 5.

1.2.2 Ordonnancement partitionné

Dans ce modèle, les tâches sont regroupées par processeur dès la phase de conception, et un algorithme d'ordonnancement local dérivé du cas monoprocesseur est employé sur chaque processeur. Les tâches sitôt affectées aux processeurs ne sont pas autorisées à migrer d'un processeur à l'autre.

L'affectation (partitionnement) des tâches aux processeurs dans cette stratégie est généralement réalisée hors-ligne sur la base des caractéristiques temporelles des tâches (facteur d'utilisation). Malheureusement trouver un partitionnement optimal est équivalent à un problème de sac à dos optimal (bin-packing) [41]. C'est donc un problème NP difficile au sens fort [79]. Avec ce modèle, gérer une panne nécessite de repartir en ligne les tâches affectées au processeur en panne sur les autres processeurs, ce qui n'est pas réaliste compte tenu de la complexité du problème. Pour le partitionnement, deux problèmes se posent selon que le nombre de processeurs est connu (fixé à l'avance) ou pas.

Dans le cas où le nombre de processeurs est connu, l'objectif est de trouver une affectation des tâches et un test d'ordonnançabilité qui permet de vérifier qu'une configuration de tâche est ordonnançable sur un nombre fixé de processeurs [83].

Dans le cas où le nombre de processeurs n'est pas fixé à l'avance et n'est pas a priori borné, les travaux de [40, 53, 92, 112] proposent des heuristiques de complexité polynomiale fondées sur une méthode d'optimisation basée sur les algorithmes First fit , Best fit Next Fit, Worst-Fit et recuit simulé. Les algorithmes basés sur ces heuristiques ont plus pour objectif de minimiser le nombre de processeurs que d'étudier en réalité l'ordonnançabilité du système. L'analyse faite permet de trouver une architecture optimale. Ces algorithmes et leur complexité sont donnés dans le tableau de la figure 1.I :

Notons tout de même que ces travaux se situent dans un cadre où le nombre de processeurs est supposé inconnu et non borné a priori, ce qui est irréaliste. Des analyses

Famille d'algorithmes	Algorithmes	Complexité
RM	RMNF [50]	$O(nlogn)$
	RMFF [50]	$O(nlogn)$
	RMBF [94]	$O(nlogn)$
	RM-FFDU [93]	$O(nlogn)$
	RMST [22]	$O(nlogn)$
	RMGT [22]	$O(nlogn)$
	RMGT/M [23]	$O(n)$
	RMNF-LL [93]	$O(n)$
	RMFF-LL [93]	$O(nlogn)$
	RMBF-LL [93]	$O(nlogn)$
Autre	FFDUF [44]	$O(nlogn)$
	RBOUND-MP [74]	$O(n(m+logn))$
EDF	EDF-NF [87]	$O(n)$
	EDF-FF [87]	$O(nlogn)$
	EDF-BF [87]	$O(nlogn)$
	EDF-WF [87]	$O(nlogn)$

Tableau 1.I – Récapitulatif de la complexité des algorithmes multiprocesseur partitionnés. n est le nombre de tâches et m le nombre de processeurs.

supplémentaires sont donc nécessaires pour prendre en compte l'ordonnançabilité. La technique d'analyse d'ordonnançabilité utilisée est basée sur l'étude du facteur d'utilisation. Les lecteurs intéressés peuvent se référer aux travaux de [3, 22, 50, 84, 86, 92, 93] pour les algorithmes à priorités fixes et aux travaux de [85, 87] pour les algorithmes à priorités dynamiques.

1.2.3 Comparaison

Du fait de la migration, l'approche globale est potentiellement plus puissante que l'approche par partitionnement d'un point de vue ordonnancement. Cependant la migration des tâches et la préemption font que l'ordonnancement global engendre un surcout processeur plus élevé lié à l'ordonnancement.
Leung et Whitehed [79] ont d'autre part montré que ces deux approches sont incomparables de façon générale. En effet sur une même plateforme matérielle, ils ont montré qu'il peut exister des systèmes de tâches qui sont ordonnançables à l'aide d'une approche

partitionnée mais qui ne le sont plus si une approche globale avec assignation de priorités fixes aux tâches est utilisée ; de même, il y a des systèmes de tâches périodiques pour lesquels il existe un ordonnancement avec approche globale avec assignation de priorités fixes aux tâches mais pour lesquelles aucun partitionnement en m sous-ensembles n'existe. Cela souligne alors l'intérêt d'étudier les deux approches dans le cas général. Dans la suite, nous considérons exclusivement l'approche globale pour les raisons précédemment évoquées.

1.3 Formalisation d'un ordonnancement

Nous considérons dans cette partie un ensemble τ de n tâches périodiques : $\tau = \{\tau_1, \ldots, \tau_n\}$.

1.3.1 Instance de tâche

Considérons une tâche $\tau_i \in \tau$ quelconque et deux instants quelconques t et t'. La tâche τ_i consiste en une suite infinie d'instances activées aux instants $r_i + k * P_i$ ($k \geq 0$). Nous introduisons les notations suivantes illustrées sur la figure 1.8 :

- $PI_i(t)$ est l'instance courante de la tâche τ_i à t (c'est la dernière instance activée à ou avant t).
- $RCT_i(t)$ est le temps d'exécution restant à l'instance $PI_i(t)$ de la tâche τ_i à l'instant t.
- $\overline{RCT_i(t)}$ correspond au temps CPU déjà consommé à l'instant t par l'instance $PI_i(t)$ de la tâche τ_i.
- $W_i(t, t')$ correspond au temps CPU reçu par la tâche τ_i entre les instants t et t'.
- $W(t, t')$ correspond au temps CPU cumulé reçu par l'ensemble des tâches entre les instants t et t'.

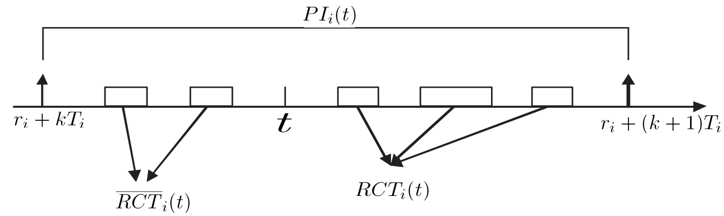


Figure 1.8 – Instance en cours d'une tâche : temps processeur reçu par l'instance en cours et temps processeur restant à exécuter par cette même instance.

Compte tenu des définitions précédentes, nous avons pour une tâche τ_i,

$$RCT_i(t) + \overline{RCT_i}(t) = C_i.$$

De plus, s'il n'y a aucun temps d'inactivité processeur entre les instants t et t', alors $W(t,t') = m(t'-t)$, où m est le nombre de processeurs de la plateforme.
Afin de pouvoir définir l'état instantané du système, nous introduisons la notion d'état d'une tâche et d'état du système.

Définition 1. *L'état d'une tâche τ_i à un instant t est défini par :*

$$st_i(t) = \begin{cases} (0, dist_i(t) = r_i - t) \;\; si \;\; \tau_i \;\; n'est \;\; pas \;\; encore \;\; active(t < r_i) \\ (RCT_i(t), dist_i(t)) \;\; sinon \end{cases}$$

où $dist_i(t)$ est la durée qui sépare l'instant t de la prochaine activation de la tâche τ_i.
L'état du système S à l'instant t est défini par $ST_S(t) = (st_1(t), st_2(t), \ldots, st_n(t))$.

Dorénavant, nous appellons *unité de temps* l'intervalle de temps $[t, t+1)$. On dit qu'une tâche a été ordonnancée à l'instant t si elle a bénéficié du processeur pendant l'unité de temps t.

1.3.2 Ordonnancement

Formellement, un *ordonnancement* sur une plateforme de m processeurs est une application O définie par :$O : \mathbb{N} \rightarrow \wp_m(\tau)$ telle que $\tau_i \in O(t) \Leftrightarrow \tau_i$ a été ordonnancée à l'instant t. Soit O_i la fonction indicatrice définie par

$$O_i(t) = \begin{cases} 1 \;\; si \;\; \tau_i \in O(t) \\ 0 \;\; sinon \end{cases}$$

Il s'en suit d'après la définition de l'état d'un système que donner un ordonnancement du système est équivalent à donner une suite d'états du système.

Un ordonnancement est *valide* si et seulement si

$$\forall \tau_i \in \tau, \sum_{t=0}^{t=r_i-1} O_i(t) = 0 \quad \text{et,}$$

$$\forall k \in \mathbb{N}, \sum_{t=0}^{t=r_i+(k-1)T_i+D_i-1} O_i(t) = \sum_{t=0}^{t=r_i+kT_i-1} O_i(t) = k * C_i.$$

Une politique d'ordonnancement est *conservative* si elle ne laisse aucun processeur inactif pendant qu'une tâche est en attente. Plus formellement, cela se définit par

$$|O(t)| < m \Rightarrow |\{i/st_i(t) = (a,b) \; avec \; a > 0\}| < m.$$

Un ordonnancement est *déterministe* si et seulement si les décisions d'ordonnancement sont les mêmes quand l'état du système est le même : $ST_S(t) = ST_S(t') \Rightarrow O(t) = O(t')$. Un ordonnancement est *cyclique* à partir d'un instant t_c et avec une périodicité T si et seulement si $\forall t \geq t_c, O(t) = O(t+T)$. Si en plus l'ordonnancement est déterministe, la propriété de cyclicité peut être déduite à partir de l'examen de l'état du système :

Lemme 1. *Soit O un ordonnancement déterministe. S'il existe un instant t_0 tel que $ST_S(t_0) = ST_S(t_0+T)$, alors O est cyclique à partir de t_0 et de périodicité T.*

Démonstration. Cela peut facilement être prouvé par induction, en utilisant l'hypothèse du déterminisme. □

Enfin, nous définissons les stratégies d'ordonnancement *monotone* comme des stratégies telles que la charge processeur restante de toute tâche τ_i à tout instant $t+T$ est supérieure ou égale à la charge processeur restante de la tâche à l'instant t.
Formellement, un *ordonnancement monotone* se définit par :

Définition 2. *Un ordonnancement déterministe conservatif O est monotone si*

$$\forall i \in 1 \ldots n, \forall t \geq r_i, RCT_i(t+T) \geq RCT_i(t).$$

Notons que cette propriété peut être reformulée en :

$$\forall i \in 1 \ldots n, \forall t \geq r_i, RCT_i(t) + \overline{RCT_i}(t+T) \leq C_i.$$

1.4 Les ordonnancements P-équitables

Dans ce paragraphe, nous nous plaçons dans le contexte des tâches indépendantes à départs simultanés et à échéances sur requêtes, dans lequel les ordonnancements P-équitables ont été définis.

1.4.1 Principe général

Dans ce modèle d'ordonnancement, il s'agit de permettre à chaque tâche de s'exécuter de façon régulière dans le temps, donc à "vitesse constante", la vitesse correspondant, pour une tâche τ_i donnée, au facteur de charge U_i de celle-ci. En effet, dans un ordonnancement idéal, chaque tâche τ_i doit recevoir exactement $U_i * t$ unités de processeur dans l'intervalle $[0,t)$. Cependant, ceci n'est pas toujours possible car le temps processeur est discret. Donc à un instant t, le nombre d'unités de temps processeur reçues par une tâche doit être entier. Les ordonnancements P-équitables tentent à chaque instant de s'approcher le plus fidèlement possible de cet ordonnancement idéal. Par conséquent on approxime la quantité $U_i * t$ soit par l'entier directement supérieur, soit par l'entier

directement inférieur. La différence entre l'ordonnancement idéal et l'ordonnancement construit est formalisée par la notion de *retard*. Formellement, le *retard* d'une tâche τ_i à un instant $t \in \mathbb{N}$ et pour un ordonnancement O noté $retard(O, \tau_i, t)$ est donné par $retard(O, \tau_i, t) = U_i * t - \sum_{j=0}^{j=t-1} O_i(t)$. Un ordonnancement est alors *P-équitable* si et seulement si

$$\forall \tau_i \in \tau, \forall t \in \mathbb{N},\ -1 < retard(O, \tau_i, t) < 1.$$

Cela veut dire que l'écart absolu entre la progression idéale et la progression réelle, pour toute tâche et à tout instant doit toujours être inférieure à l'unité.
Ceci implique qu'à chaque instant t chaque tâche τ_i de τ doit avoir reçu $\lfloor U_i * t \rfloor$ ou $\lceil U_i * t \rceil$ unités de temps processeur.
Graphiquement cela s'interprète de la façon suivante : si nous notons $W(t)$ la charge processeur idéale reçue par une tâche, lors d'un ordonnancement P-équitable, la courbe représentant la charge processeur réellement reçue par la tâche est comprise entre les droites d'équation $W_{-} = U_i * t - 1$ et $W^{+}(t) = U_i * t + 1$. La charge processeur reçue étant une valeur discrète, la courbe représentant cette charge processeur reçue est une ligne brisée. La tâche est en avance si cette ligne brisée est au dessus de la droite $W(t)$ mais en dessous de $W^{+}(t) = U_i * t + 1$ et en retard si la ligne est en dessous de la droite $W(t)$ mais au dessus de $W_{-} = U_i * t - 1$. Ceci est illustré par la figure 1.9.

Avant de présenter la mise en œuvre des ordonnancements P-équitables ainsi que les algorithmes *PF*, *PD* et PD^2 dans les sections à venir, nous allons présenter quelques notions qui leurs sont communes :

– La chaîne caractéristique d'une tâche τ_i à un instant $t \in \mathbb{N}$ notée $\alpha_t(\tau_i)$ est une suite infinie des caractères $\{-, 0, +\}$ avec

$$\alpha_t(\tau_i) = sign(U_i * (t+1) - \lfloor U_i * t \rfloor - 1).$$

– La sous chaîne caractéristique d'une tâche τ_i à un instant $t \in \mathbb{N}$ notée $\alpha(\tau_i, t)$ est une chaîne finie définie par :

$$\alpha(\tau_i, t) = \alpha_{t+1}(\tau_i)\alpha_{t+2}(\tau_i)\ldots\alpha_{t'}(\tau_i)$$

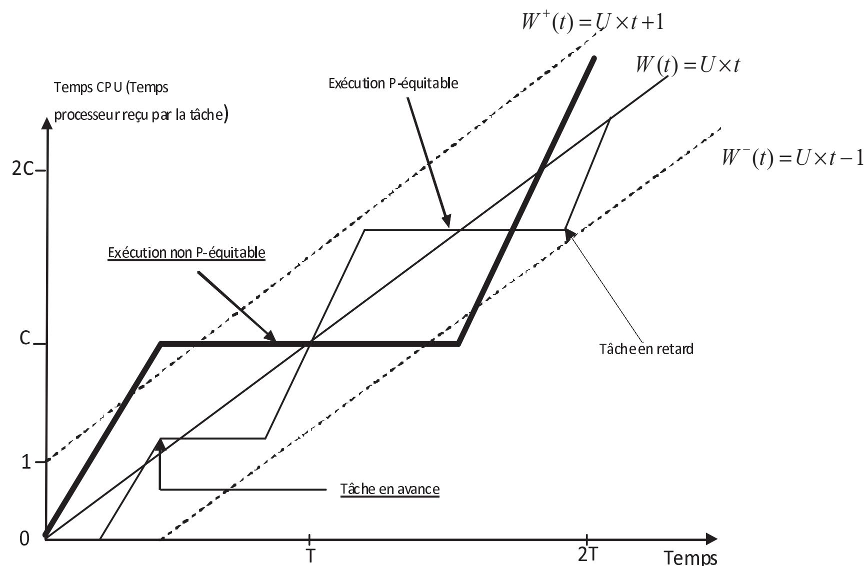


Figure 1.9 – P-équité

avec $t' = min\{j : j > t \text{ et } \alpha_j(\tau_i) = 0\}$.

- Dans un contexte d'ordonnancement P-équitable O et à un instant t, une tâche τ_i est dite *en avance* si et seulement si $retard(O, \tau_i, t) < 0$, *en retard* si et seulement si $retard(S, \tau_i, t) > 0$, *ponctuelle* si et seulement si elle n'est ni en avance ni en retard.
- Dans un contexte d'ordonnancement P-équitable O et à un instant t, une tâche τ_i est dite *interdite* si et seulement si elle est en avance et $\alpha_t(\tau_i) \neq +$; elle est dite *urgente* si et seulement si elle est en retard et $\alpha_t(\tau_i) \neq -$; enfin elle est *possible* si et seulement si elle n'est ni interdite ni urgente.

De manière informelle ces notions s'interprètent de la façon suivante : à tout instant t,

- une tâche *urgente* est une tâche qui est en retard et qui serait sous la droite limite inférieure à l'instant t si le processeur ne lui était pas attribué. Cette tâche doit donc bénéficier du processeur à cet instant car sinon l'équité sera violée.
- une tâche *interdite* est une tâche qui est en *avance* et qui serait au dessus de la droite limite supérieure à l'instant t si le processeur lui était attribué. Cette tâche ne doit donc pas bénéficier du processeur à cet instant car sinon l'équité ne serait

plus respectée.
- une tâche *possible* n'est ni urgente ni interdite et l'équité ne sera pas violée que le processeur lui soit attribué ou non.

Pour les trois algorithmes, mis à part le traitement des cas d'ex-æquo l'algorithme d'élection des tâches les plus prioritaires est le suivant : pour une configuration de n tâches périodiques à départs simultanés à échéances sur requête $\tau = \{\tau_1, \tau_2, \dots \tau_n\}$ et sur une plateforme de m processeurs identiques, la politique d'élection des tâche par les algorithmes P-équitables peut se résumer ainsi :

1. déterminer les sous ensembles des tâches urgentes, possibles et interdites ;
2. allouer les tâches urgentes (dont le nombre est inférieur ou égal à m) aux processeurs ;
3. allouer aux processeurs libres les tâches possibles les plus prioritaires.

1.4.2 Mise en oeuvre : les sous-tâches

Pour des raisons d'implémentation, chaque tâche τ_i est subdivisée en sous-tâches de durée unitaire. Afin d'assurer la régularité de l'exécution de la tâche, les sous-tâches doivent s'exécuter dans des intervalles appelés *fenêtres d'exécution*. Différentes sous-tâches d'une tâche peuvent s'exécuter sur des processeurs différents (car la migration est autorisée) mais pas simultanément (c.-à-d. que le parallélisme au sein d'une tâche est interdit).

La $j^{ième}$ sous tâche de la tâche τ_i avec $j \geq 0$ est notée τ_i^j. Chaque sous-tâche τ_i^j a une *pseudo date d'activation* r_i^j et une *pseudo échéance* d_i^j définis par : $r_i^j = \left\lfloor \frac{j}{U_i} \right\rfloor$ et $d_i^j = \left\lceil \frac{j+1}{U_i} \right\rceil$.

La fenêtre d'exécution de la sous-tâche τ_i^j est alors $\left[r_i^j, d_i^j\right)$ et la taille de la fenêtre notée $|I_i^l|$ vaut $d_i^j - r_i^j$.

Remarquons que r_i^{j+1} coïncide avec $d_i^j - 1$ ou d_i^j. Cela veut dire que deux sous tâches consécutives de la même tâche ont des fenêtres d'exécution qui soit sont disjointes, soit se chevauchent sur une unité de temps. L'exemple de la figure 1.10 illustre une exécution

P-équitable de la tâche $\tau_i(0,3,5,5)$.

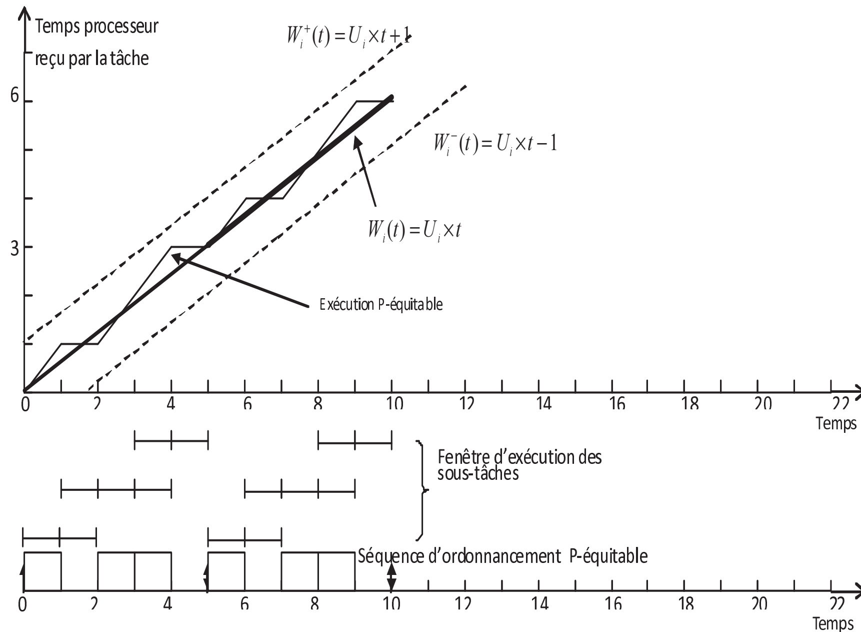


Figure 1.10 – Ordonnancement P-équitable de la tâche $\tau_i = (0,3,5,5)$.
Les fenêtres d'exécution des sous-tâches sont aussi illustrées.Ici, $b_i^1 = 1$ *et* $b_i^3 = 0$

La valeur $b_i^j = b(\tau_i^j)$ définie par $b_i^j = d_i^j - r_i^{j+1} = \left\lceil \frac{j+1}{U_i} \right\rceil - \left\lfloor \frac{j+1}{U_i} \right\rfloor$, est appelée *bit successeur* de la sous tâche τ_i^j. Le bit successeur d'une sous tâche τ_i^j ne peut prendre que deux valeurs : 0 et 1. Ce bit successeur vaut 0 si la fenêtre d'exécution de la sous tâche τ_i^j est disjointe de celle de la prochaine sous tâche de la tâche τ_i, c'est à dire τ_i^{j+1}. Et le bit successeur de la sous tâche τ_i^j vaut 1 si les deux fenêtres se chevauchent d'une unité de temps.

L'idée ayant conduit à l'utilisation de la notion de bit de successeur de manière générale est d'éviter un effet domino qui caractérise le fait que l'ordonnancement d'une sous tâche de bit successeur 1 sur la dernière unité de temps de sa fenêtre temporelle entraîne nécessairement un décalage de la sous-tâche suivante et ce récursivement.

1.4.3 Les algorithmes P-équitables

En ce qui concerne l'ordonnançabilité, à ce jour, trois algorithmes d'ordonnancement P-équitables optimaux ont été proposés. Il s'agit de *PF* [18], *PD* [17], PD^2 [5]. Ces algorithme sont une adaptation de l'algorithme *EPDF* [4] qui est un algorithme *sous-optimal* dans ce contexte. L'algorithme *EPDF*, pour Earliest-Pseudo-Deadline-First, attribue la priorité aux sous tâches selon une politique *EDF*. Malheureusement, Srinivasan et Anderson ont prouvé dans [4, 105] que *EPDF* seul ne peut être optimal que dans un contexte biprocesseur. Cependant *EPDF* peut être rendu optimal dans le cas où $m \geq 2$ en lui associant des règle de gestion des ambiguïtés (cas d'égalité de priorité entre sous tâches). Les algorithmes *PF*, *PD*, $etPD^2$ ont donc été obtenus en ajoutant des règles de gestion des ambiguïtés à l'algorithme *EPDF*. Ces trois algorithmes diffèrent dans la manière de résoudre les problèmes de conflits.

1.4.4 L'algorithme PF

L'algorithme d'ordonnancement *PF* est le premier algorithme obtenu en ajoutant des règles de gestion des ambiguïtés à *EPDF* et c'est le premier algorithme à avoir été prouvé optimal [18]. Sous l'algorithme d'ordonnancement PF, chaque tâche est subdivisée en sous-tâches et les priorités sont attribuées aux sous-tâches selon l'algorithme suivant :
à un instant t, si les sous-tâche τ_i^u et τ_j^v sont prêtes, alors τ_i^u est plus prioritaire que τ_j^v, ce que l'on note $\tau_i^u \succ \tau_j^v$, si l'une des conditions suivante est vérifiée :

- $d_i^u < d_j^v$ (règle 1) ;
- $d_i^u = d_j^v$ et $b_i^u > b_j^v$ (règle 2) ;
- $d_i^u = d_j^v$, $b_i^u = b_j^v = 1$ et $\tau_i^{u+1} \succ \tau_j^{v+1}$ (règle 3).

Si des sous-tâches n'ont pas pu être départagées par les conditions ci-dessus, alors le choix de la sous-tâche la plus prioritaire se fait de manière arbitraire.
Avec PF, l'ambiguïté est donc levée en comparant les bits successeurs (règle deux (2)) , puis un appel récursif (règle trois (3)) aux règles une (1) et deux (2) appliquées aux sous tâches suivantes. Cette récursivité de la règle trois (3) rend cet algorithme plus complexe.

Theorème 1. Optimalité de *PF* ([18]) *: l'algorithme d'ordonnancement PF est optimal pour l'ordonnancement de tâches périodiques indépendantes à départs simultanés et à échéances sur requêtes sur une plateforme multiprocesseur. Dans ce contexte, un système de tâches est ordonnançable si et seulement si $U \leq m$.*

1.4.5 L'algorithme PD

Baruah, Gehrke et Plaxton ont proposé plus tard dans [17] l'algorithme *PD* qui simplifie *PF* en éliminant l'appel récursif par remplacement de la règle 3 de PF. Avant de présenter l'algorithme *PD* [17], nous allons introduire la notion d'*échéance de groupe*. Tout d'abord, nous dirons d'une tâche τ_i dans la suite qu'elle est de *poids lourds* si $U_i \geq \frac{1}{2}$. Elle est de poids léger si $U_i < \frac{1}{2}$. Il a été montré dans [5], qu'une tâches τ_i est de *poids lourds* si et seulement si $\left|I_i^1\right| = 2$ et $\forall j \geq 2, \left|I_i^j\right| = 2$ *ou* 3.
Soit $\tau_i^j, \ldots, \tau_i^l$ une suite de sous tâches, dans cet ordre, de la tâche τ_i de poids lourd telle que :

$$b_i^j = 1 \; et \; (\left|I_i^k\right| = 2 \; et \; b_i^k = 1) \; et \; \left(\left|I_i^{l+1}\right| = 3 \;\; ou \; (\left|I_i^{l+1}\right| = 2 \; et \; b_i^{l+1} = 0)\right)$$

avec $j < k \leq l$.
Si une sous tâche de l'ensemble $\tau_i^j, \ldots, \tau_i^l$ venait à être ordonnancée sur la dernière unité de temps de sa fenêtre d'exécution, alors les sous tâches suivantes de cet ensemble devraient être ordonnancées sur la dernière unité de temps de leur fenêtre de réalisation. En réalité, l'ensemble $\tau_i^j, \ldots, \tau_i^l$ peut être considéré comme une seule et même entité à être ordonnancée et soumise à une *échéance de groupe*. Formellement, l'échéance de groupe d'une groupe de $\tau_i^j, \ldots, \tau_i^l$ vaut :

$$\begin{cases} d_i^l + 1 \; si \; \left|I_i^{l+1}\right| = 3 \\ d_i^l \; si \; b_i^l = 0 \end{cases}$$

Intuitivement, si l'on décompose une tâche de poids lourd en sous tâches, et on se met dans la situation où toutes les sous tâches s'exécutent sur la première unité de temps de leur fenêtre de réalisation, alors les instants d'inactivité de cette tâche correspondent aux

échéances de groupe de cette tâche. Plus précisément, les échéances de groupes repèrent la fin d'une unité de temps d'inactivité. Généralement, soit τ_i une tâche et τ_i^j une sous tâche de τ_i, alors l'échéance de groupe de τ_i^j notée D_i^j est définie par :

$$D_i^j = \begin{cases} Min\left\{u\ t.q \text{ u est une échéance de groupe de } \tau_i\ et\ u \geq d_i^j\right\}\ si\ U_i \geq \frac{1}{2} \\ 0\ si\ U_i < \frac{1}{2} \end{cases}$$

De même, pour une sous tâche donnée τ_i^j, on définit son *bit successeur* B_i^j de groupe comme étant le bit successeur de la dernière sous tâche du groupe auquel elle appartient. Intuitivement, la notion de bit successeur de groupe a été introduite pour différencier les groupes de sous tâches dont la fenêtre d'exécution de la dernière sous tâche est de taille 3 (dans ce cas le bit successeur du groupe est égal à 1) de ceux dont la dernière fenêtre d'exécution est de taille 2 (dans ce cas le bit successeur du groupe est égal à 0).
Maintenant que les notion d'échéance et de bit successeur de groupe sont définies, nous allons présenter l'algorithme *PD*. Les tâches étant décomposées en sous tâches, la sous tâche τ_i^u de la tâche τ_i est plus prioritaire que la sous tâche τ_j^v de la tâche τ_j si :

- $d_i^u < d_j^v$;
- $d_i^u = d_j^v$ et $b_i^u > b_j^v$;
- $d_i^u = d_j^v$ et $b_i^u = b_j^v = 1$ et $D_i^u > D_j^v$;
- $d_i^u = d_j^v$ et $b_i^u = b_j^v = 1$ et $D_i^u = D_j^v$ et $U_i > U_j$;
- $d_i^u = d_j^v$ et $b_i^u = b_j^v = 1$ et $D_i^u = D_j^v$ et $U_i = U_j$ et $B_i^u > B_j^v$;

Il peut arriver que des sous-tâches ne soient pas départagées par les conditions ci-dessus, alors le choix de la sous-tâche le plus prioritaire se fait de manière arbitraire.

Theorème 2. Optimalité de *PD* ([17]) *: l'algorithme d'ordonnancement PD est optimal pour l'ordonnancement de tâches périodiques indépendantes à départs simultanés et à échéances sur requêtes sur une plateforme multiprocesseur. Dans ce contexte, un système de tâches est ordonnançable si et seulement si* $U \leq m$.

1.4.6 L'algorithme PD^2

Anderson et Srinivasan dans [5], ont montré que les règles quatre (4) et cinq (5) de l'algorithme PD étaient de trop. Ils ont alors proposé l'algorithme PD^2 [5, 6] qui est en fait l'algorithme PD débarrassé de ses règles quatre (4) et cinq (5).
Dans un contexte d'ordonnancement avec PD^2, les tâches sont décomposées en sous tâches. La sous tâche τ_i^u de la tâche τ_i est plus prioritaire que la sous tâche τ_j^v de la tâche τ_j si :

- $d_i^u < d_j^v$;
- $d_i^u = d_j^v$ et $b_i^u > b_j^v$;
- $d_i^u = d_j^v$ et $b_i^u = b_j^v = 1$ et $D_i^u > D_j^v$;

Dans ce cas aussi il peut arriver que des sous-tâches ne soient pas départagées par les conditions ci-dessus, alors le choix de la sous-tâche le plus prioritaire se fait de manière arbitraire.

Theorème 3. Optimalité de PD^2 ([5]) *: l'algorithme d'ordonnancement PD^2 est optimal pour l'ordonnancement de tâches périodiques indépendantes à départs simultanés et à échéances sur requêtes sur une plateforme multiprocesseur. Dans ce contexte, un système de tâches est ordonnançable si et seulement si $U \leq m$.*

L'optimalité de PD^2 [5] montre alors que les deux dernière conditions de l'algorithme PD était en réalité inutiles. Dans tous les cas on peut remarquer que toutes les valeurs dont on a besoin pour lever une ambiguïté peuvent être calculées hors-ligne rendant ainsi la mise en oeuvre de ces algorithmes plus faciles que la mise en oeuvre de PF. D'autre part, des deux derniers algorithmes, PD^2 est celui qui a le moins de conditions donc le plus facile à mettre en oeuvre, ce qui fait dire à [7] que PD^2 est le meilleur des trois algorithmes P-équitables connus optimaux.

Il y a eu dans la littérature une version statique de l'ordonnancement P-équitable. A notre connaissance le seul algorithme P-équitable attribuant les priorités de manière statique est l'algorithme WM proposé par Baruah dans [16]. Cet algorithme attribue les

priorités aux tâches proportionnellement à leur poids. Il est montré dans [16] que l'algorithme WM est optimal dans un contexte monoprocesseur pour un système de tâches périodiques à échéances sur requêtes avec $U \leq ln2$. Les travaux de Ramamurty et Moir dans [99] montrent que le critère d'ordonnançabilité donné dans [16] est une condition suffisante d'ordonnançabilité d'un système de tâches indépendantes à échéances sur requête par WM. Anderson et Johnson utilisent les travaux de [99] pour prouver dans [3] que la condition $U \leq \frac{m}{2}$ est un critère analytique d'ordonnançabilité avec WM en multiprocesseur. Ils prouvent par la suite qu'aucune autre politique d'ordonnancement à priorités statiques ne peut fournir un meilleur critère [3].
L'utilisation d'un algorithme P-équitable avec assignation statique des priorités fait perdre alors la moitié de la capacité des processeurs globalement comparativement aux algorithmes P-équitables optimaux.

1.5 Conclusion

Nous avons présenté dans ce chapitre les définitions formelles liées à la notion d'ordonnancement et les ordonnancements P-équitables. Nous avons présenté les différents algorithmes d'ordonnancements P-équitables qui existent ainsi que leur résultat d'optimalité. Ces différents algorithmes ne diffèrent les uns des autres que par leur façon de départager les ex-æquo.

CHAPITRE 2

ORDONNANCEMENT HORS-LIGNE

Lorsque des contraintes plus complexes sont à prendre en compte (ressources, précédence, etc.), peu de résultats existent concernant les algorithmes en-ligne et il n'existe pas d'algorithme optimal. Une solution alternative pour ordonnancer de tels systèmes est de faire alors appel aux techniques hors-ligne. Un autre intérêt des techniques d'ordonnancement hors-ligne est de permettre la détermination de séquences d'ordonnancements valides vérifiant des contraintes qualitatives additionnelles telles que l'optimisation du temps de réponse de certaines tâches ou la minimisation de la gigue d'autres tâches. Ces critères ne sont pas considérés par les algorithmes en-ligne classiques.
Les stratégies d'ordonnancement hors-ligne sont les seules stratégies qui permettent d'obtenir un ordonnancement optimal dans le cas général et de traiter certaines configurations que les algorithmes en-ligne ne savent pas ordonnancer. Ces stratégies ont donc une puissance d'ordonnancement supérieure. En effet, la connaissance a priori des dates d'activation de toutes les instances de toutes les tâches (on parle de clairvoyance) les rend plus performantes que les stratégies d'ordonnancement en-ligne qui fondent leurs décisions sur la seule connaissance instantanée du système. Cependant, la mise en œuvre de telles stratégies est nettement plus coûteuse, notamment parce qu'il s'agit souvent de stratégies d'exploration exhaustive de l'espace des solutions.

2.1 Principe des approches à base de modèle

2.1.1 Schéma général

Les approches à base de modèle sont des approches permettant l'exploration exhaustive de l'espace d'état du système. Ce sont des approches énumératives. Elles permettent de vérifier l'ordonnançabilité du système avant son lancement. Une fois l'ordonnançabilité de l'application établie, une séquence d'ordonnancement est calculée et implantée au sein d'un séquenceur pour être répétée à l'infinie.

D'autre part, ces approches permettant une exploration exhaustive de l'espace d'état du système, elles permettent de vérifier de nombreuses propriétés du système autre que l'ordonnançabilité. Elles permettent par conséquent d'affiner le choix de la stratégie à utiliser et donc d'optimiser la stratégie.

Cependant, ces approches s'appuyant sur une construction exhaustive de l'espace des solutions afin d'extraire une séquence peuvent conduire à une problème d'explosion combinatoire pour les systèmes complexes. De plus, dès qu'une séquence est extraite et utilisée, si on veut faire évoluer l'ensemble des tâches, il faut recalculer une nouvelle séquence. Cela empêche alors par exemple l'adjonction ou la suppression en ligne de tâches.

Compte tenu de la complexité d'utilisation de ces approches et de leur rigidité, elles sont réservées aux cas où les paradigmes en lignes ne s'appliquent pas. Mais s'il y a beaucoup d'interactions, l'espace des solutions sera plus maitrisable et on peut ajouter des heuristiques pour accélérer les calculs.

2.1.2 Approches présentes dans la littérature

Les méthodes d'ordonnancement hors-ligne fondent leur stratégie sur la modélisation de l'application par des systèmes à états (automates, réseaux de Petri etc.). Certains sont à temps explicite [2, 54, 100] (on parle d'automate temporisé), d'autres à temps implicite [36, 63]. Les modèles à temps explicite sont exploités via des techniques de simulation, ceux à temps implicite via des techniques d'analyse. Dans les deux cas, un premier axe d'étude consiste à produire des techniques de modélisation exhaustives i.e. qui prennent en compte l'ensemble des configurations réelles possibles.

Les stratégies d'ordonnancement hors-ligne connues dans la littérature traitent soit de systèmes de tâches non périodiques, soit imposent des hypothèses restrictives. Certaines stratégies ont également pris en compte des tâches liées entre elles par des contraintes de précédence ou de ressources. Les travaux de [97] ont considéré le partage de ressources, et proposé des algorithmes basés sur une exploration arborescente de l'espace des solutions, couplée avec l'utilisation d'heuristiques permettant de limiter le nombre de branches explorées. Les travaux de [56] ont pris en compte le problème de la pré-

cédence dans le cas d'une architecture biprocesseur. Dans leurs travaux, [56] opèrent par réajustements successifs de certains des paramètres temporels. Quant à [91], leurs travaux ont étudié le problème de l'ordonnancement en présence de contraintes de précédence, pour un nombre quelconque de processeurs, mais en l'absence d'échéances. Enfin, les travaux de [116] présentent une stratégie permettant de prendre en compte à la fois le partage de ressources et les contraintes de précédence. Le principe consiste à utiliser une exploration arborescente, couplée avec des étapes de découpages des tâches, de modifications des paramètres et d'adjonction de contraintes complémentaires (permettant la gestion des sections critiques).

2.2 Les approches à base de réseaux de Petri

2.2.1 Les réseaux de Petri

Le lecteur intéressé par les définitions de base sur les réseaux de Petri peut se référer à [34].

Définition 3. *Réseau de Petri : un réseau de Petri est un triplet $N =< P,T,W >$ où :*

- $P = \{p_1,\dots,p_n\}$ *est un ensemble fini de places,*
- $T = \{t_1,\dots,t_m\}$ *est un ensemble fini de transitions,*
- $W : P \times T \cup T \times P \longrightarrow \mathbb{N}$ *,* $\mathbb{N}$ *est l'ensemble des entiers naturels, est une fonction de pondération (valuation) des arcs*

Un réseau de Petri est un graphe bipartite orienté. Ce graphe possède deux sortes de sommets, les *places* et les *transitions*. Les arcs sont pondérés et relient une place à une transition ou une transition à une place. On convient que $W(p,t) = 0$ (respectivement $W(t, p) = 0$) lorsqu'il n'y a pas d'arc de p vers t (respectivement de t vers p).
Les places d'un réseau contiennent des *jetons* dont la variation du nombre permet de décrire la dynamique du réseaux.

Définition 4. *Un marquage M est une fonction $M : P \longrightarrow \mathbb{N}$ qui décrit la répartition des jetons. Un réseau initialisé est un réseau de Petri N assorti d'un marquage M_0 dit*

marquage initial. On le note (N,M_0).

Définition 5. *Une transition* $t \in T$ *est franchissable (ou sensibilisée) à partir d'un marquage* M *si et seulement si* $\forall p \in P, M(p) \geq W(p,t)$.
Lorsque la transition t *est franchissable à partir du marquage* M*, on définit le tir de la transition* t *comme l'action qui modifie le marquage* M *en un marquage* M' *défini pour toute place* $p \in P$ *par* $M'(p) = M(p) - W(p,t) + W(t,p)$.
On note $M[t\rangle$ *lorsque* t *est franchissable depuis le marquage* M *et* $M[t\rangle M'$ *lorsque* M' *est le résultat du tir de* t *depuis* M.
On étend cette notation à une séquence de tirs $u \in T^*$ *avec* $u = t_1.t_2.\ldots.t_n$ *en posant* $M[u\rangle M'$ *lorsqu'il existe une suite de marquages* $M_1,\ldots,M_{n-1}$ *telle que*

$$M[t_1\rangle M_1 [t_2\rangle M_2 \ldots [t_{n-1}\rangle M_{n-1} [t_n\rangle M'.$$

On dit que u *est une* séquence franchissable *depuis* M.
La relation $[.\rangle$ *entre les marquages de* N *est appelée relation de tir de* N

On dit d'un marquage M de N qu'il est *accessible* depuis le marquage initial M_0 lorsqu'il existe une séquence de tir u telle que $M_0[u\rangle M$.

Définition 6. *Une place est bornée si pour tout marquage accessible, le nombre de jetons qu'elle contient est borné.*
Un réseau de Petri est borné si toutes ses places sont bornées.

Les comportements d'un réseau de Petri sont définis par ses séquences de tir. Il existe différentes structures pour représenter ces comportements, parmi lesquelles *le graphe des marquages*.

Définition 7. *Le graphe des marquages du réseau initialisé* (N,M_0) *est le graphe orienté et étiqueté noté* $G(N,M_0)$ *dont les sommets sont les marquages de* N *accessibles depuis* M_0 *et dont les arcs sont définis par la relation de tir de* N *: il existe un arc de* M *à* M' *étiqueté par* t *si et seulement si* $M[t\rangle M'$.

Définition 8. *Le langage d'un réseau initialisé* (N,M_0) *est le langage* $L(N,M_0) \subseteq T^*$ *défini par* $L(N,M_0) = L(G(N,M_0))$

Dans le comportement d'un réseau de Petri tel que nous l'avons défini, les actions (étiquettes des transitions pour le graphe des marquages, éléments de l'alphabet pour le langage) sont les transitions du réseau. Cependant, une définition plus générale utilise une fonction d'étiquetage sur un alphabet d'actions fini Σ.

Définition 9. *Un réseau de Petri étiqueté sur un alphabet* Σ *est un couple* (N, M_0, σ) *où* $N =< P, T, F, W >$ *est un réseau de Petri et* $\sigma : T \longrightarrow \Sigma$ *est une fonction d'étiquetage des transitions de* N.

La fonction σ est étendue à $\sigma : T^* \longrightarrow \Sigma^*$ par $\sigma(a_1 \dots a_k) = \sigma(a_1) \dots \sigma(a_k)$. Nous redéfinissons alors la notion de graphe des marquages et de langage en prenant en compte l'étiquetage.

Définition 10. *Le graphe des marquages du réseau étiqueté* (N, M_0, σ) *est le graphe orienté et étiqueté noté* $G(N, M_0, \sigma)$ *dont les sommets sont les marquages de* N *accessibles depuis* M_0 *et dont les arcs sont définis par la relation de tir de* N *: il existe un arc de* M *à* M' *étiqueté par* $\sigma(t)$ *si et seulement si* $M[t\rangle M'$.

Définition 11. *Le langage étiqueté d'un réseau initialisé* (N, σ, M_0) *est le langage* $L(N, \sigma, M_0) \subseteq \Sigma^*$ *défini par* $L(N, \sigma, M_0) = L(G(N, \sigma, M_0))$.

$$L(N, \sigma, M_0) = \left\{ \omega | \exists M \in \mathbb{N}^{|P|}, t_a, t_b, \dots, \in T, M_0 [t_a t_b \dots\rangle M,\ \omega = \sigma(t_a t_b \dots) \right\}.$$

Le langage des réseaux de Petri représente donc l'ensemble des séquences franchissables à partir d'un marquage initial. Il peut cependant être intéressant de ne considérer que les séquences de transitions qui mènent à un ensemble de marquages particuliers (cela peut correspondre aux marquages terminaux qui représentent les états terminaux du système) ou à un sous ensemble du langage du réseau (ce sous ensemble peut regrouper des mots tels que les états atteints font tous partie d'un ensemble de marquages que l'on juge plus intéressants que d'autres). Les notions de *langage terminal* et de *centre du langage terminal* que nous introduisons ci-dessous permettent de modéliser cela.

Définition 12. *Le langage terminal d'un réseau de Petri étiqueté est le langage de ce réseau, restreint aux séquences de transitions qui mènent à un marquage de l'ensemble*

terminal. Il est noté $L(N,\sigma,M_0,\Psi)$ où Ψ est ensemble de marquages terminaux .

$$L(N,\sigma,M_0,\Psi)=\{\omega|\exists M\in\Psi,t_a,t_b\ldots\in T,M_0\,[t_at_b\ldots\rangle\,M,\omega=\sigma(t_at_b\ldots)\}\,.$$

$\Psi\subseteq\mathbb{N}^{|P|}$ est l'ensemble des *marquages terminaux*, généralement donné sous forme de contraintes logiques sur les marquages, on l'appelle donc aussi ensemble des *contraintes terminales*.

Définition 13. *Le centre $L_c(N,\sigma,M_0,\Psi)$ du langage terminal $L(N,\sigma,M_0,\Psi)$ d'un réseau de Petri (N,σ,M_0) est le sous ensemble du langage terminal défini par*

$$L(N,\sigma,M_0,\Psi)=\{\omega|\exists M_1,M_2,\ldots,M_k\in\Psi,t_a,t_b,\ldots,t_k\in T,M_0\,[t_a\rangle\,M_1\,[t_b\rangle\,M_2\ldots[t_k\rangle\,M_k\}\,.$$

Malgré tout leur intérêt, les réseaux de Petri trouvent rapidement des limites lorsque la complexité des systèmes à étudier augmente ; en effet, c'est la représentation graphique du réseau de Petri qui lui confère ses caractéristiques les plus intéressantes. Les *réseaux de Petri colorés* introduits par [69], permettent d'alléger la représentation graphique de grands modèles complexes à partir d'un nombre très limité de places puisque la coloration des jetons permet de regrouper plusieurs places en une seule. Nous présentons ici une version très simplifiée des réseaux colorés.

Définition 14. *Réseau de Petri coloré : un réseau de Petri est un quadruplet $N=<P,T,C,W>$ où :*

- *$P=\{p_1,\ldots,p_n\}$ est un ensemble fini de places,*
- *$T=\{t_1,\ldots,t_m\}$ est un ensemble fini de transitions, flot,*
- *$C=\{c_1,\ldots,c_k\}$ est un ensemble fini de couleurs,*
- *$W:(P\times T\cup T\times P)\times C\longrightarrow\mathbb{N}$ est une fonction de pondération des arcs. On note $W(p_i,t_i,c_i)$ la composante de couleur $c_i\in C$ du poids de l'arc (p_i,t_i).*

Un *marquage coloré M* est une fonction $M:P\times C\longrightarrow\mathbb{N}$ qui décrit la répartition des jetons par couleur. Le marquage est donc représenté par une matrice avec une colonne par couleur. $M(p_i,c_i)$ est la composante de couleur $c_i\in C$ du marquage de la place $p_i\in P$. Le tir des transitions s'exprime alors de la façon suivante :

Définition 15. *Tir de transitions d'un réseau coloré : une transition $t \in T$ est franchissable (ou sensibilisée) à partir d'un marquage M si et seulement si*
$\forall p \in P, \forall c_k \in C, M(p_i, c_k) \geq W(p_i, t, c_k)$. *Lorsque la transition t est franchissable à partir du marquage M, on définit le tir de la transition t par l'action qui modifie le marquage M en un marquage M′ défini pour toute place $p_i \in P$ et pour toute couleur $c_k \in C$ par* $M'(p_i, c_k) = M(p_i, c_k) - W(p_i, t, c_k) + W(t, p_i, c_k)$.

Les définitions précédentes correspondent à une sémantique de l'entrelacement : les actions s'exécutent une à une, il n'y a pas de parallélisme effectif, on représente seulement le pseudo-parallélisme. Une alternative consiste à considérer la sémantique du parallélisme maximal. Celle-ci s'exprime grâce à la règle du tir maximal.

Définition 16. *La règle de tir maximal : soit $F = \{t_{i_1}, t_{i_2}, \ldots, t_{i_k}\}$ l'ensemble des transitions franchissables (dont certaines peuvent être en conflit), alors les ensembles de transitions à franchir simultanément $f \subseteq F$ sont tels que pour toute transition $t_i \in F - f$, t_i est en conflit avec au moins une transition de f et deux transitions distinctes quelconques de f ne sont pas en conflit.*

L'avantage de l'utilisation des réseaux de Petri autonomes fonctionnant sous la règle de tir maximal est leur simplicité de représentation et d'implémentation. Le lecteur intéressé peut se référer à [68, 111] pour une étude des propriétés de ce modèle.

2.2.2 Modélisation d'une application temps-réel à base de réseaux de Petri : sémantique et prise en compte des contraintes temporelles

Dans cette section nous présentons la modélisation de systèmes de tâches temps réel à l'aide de réseaux de Petri colorés avec marquages terminaux fonctionnant sous la règle de tir maximal. Notre travail est une extension des travaux présentés dans [60–64]. Dans cette section, $\mathbb{P}$ désigne l'ensemble des places du réseau , $\mathbb{T}$ l'ensemble des transitions et M la fonction de marquage définie par : $M : \mathbb{P} \longrightarrow \mathbb{N} \cup \{a\} \cup \{b\}$ où $\{a\}$ et $\{b\}$ sont des couleurs de jetons. La modélisation d'une application se décompose en deux parties

(voire la figure 2.1) : une partie représentant la structure temporelle et l'autre partie représentant le système de tâches.

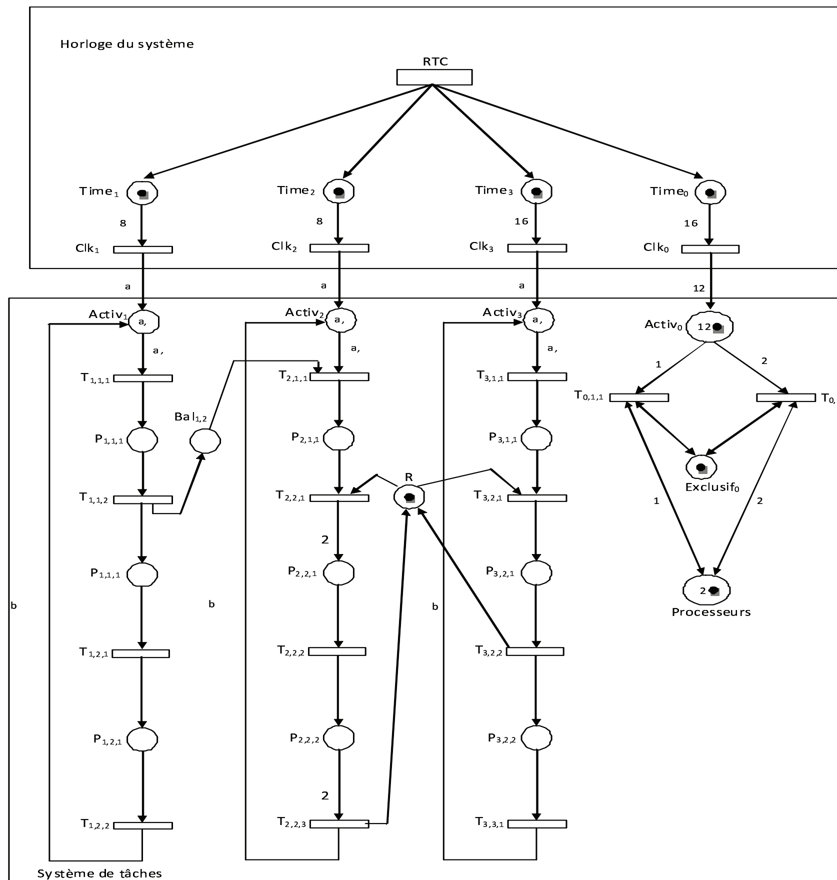


Figure 2.1 – Modélisation d'une application :
cette figure illustre la modélisation du système de tâches $\tau = \{\tau_1, \tau_2, \tau_2\}$ *décrit par le tableau 2.I.*

2.2.2.1 La structure temporelle

Cette partie contient comme l'indique la figure 2.1 :

– *l'horloge globale* notée *RTC (Real Time Clock)*. Cette transition est tirée à chaque tir de transition (à cause de la règle de tir maximal). Ceci nous permet de modéliser

Tâche $\tau_1 = (0,4,8,8)$	Tâche $\tau_2 = (0,5,8,5)$	Tâche $\tau_3 = (0,4,8,8)$
Début $b_{1,1} = 2$; Déposer_BAL(MB) ; $b_{1,2} = 2$; Fin ;	Début Retirer_BAL(MB) ; $b_{2,1} = 1$; P(R,1) ; $b_{2,2} = 4$; V(R,1) ; Fin ;	Début $b_{3,1} = 1$; P(R,1) ; $b_{3,2} = 2$; V(R,1) ; $b_{3,3} = 1$; Fin ;

Tableau 2.I – Système de tâche $\tau = \{\tau_1(0,4,8,8); \tau_2(0,5,8,8); \tau_3(0,4,8,8)\}$

le temps. Un tir de RTC modélise une unité de temps. A chaque tir, un jeton est déposé dans chacune des places composant les horloges locales.

– *les horloges locales* des tâches qui permettent de réactiver les tâches à chaque fin de période de celle-ci. L'horloge locale d'une tâche τ_i, comme l'illustre la figure 2.2 est composée d'une place accumulatrice de temps $Time_i$, qui reçoit à chaque unité de temps un jeton de l'horloge globale RTC. Lorsqu'elle contient T_i jetons, la transition Clk_i est tirée et elle dépose un jeton a dans la première place $Activ_i$ de la partie système de tâches de la tâche τ_i.

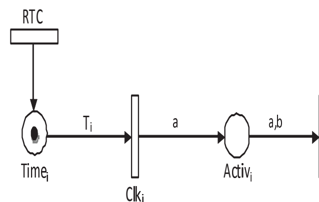


Figure 2.2 – Modélisation de l'horloge de la tâche τ_i

2.2.2.2 Le système de tâches périodiques

Dans cette partie du réseau, chaque transition est associée à une action (voir la figure 2.3). Il y a une place processeur qui contient autant de jetons que le système comporte de processeurs. Les communications sont représentées à l'aide de places de type boite

aux lettres, et il y a une place par ressource. Dans la suite de cette section on notons $\Re = \{R_1, ..., R_{|\Re|}\}$ les ressources critiques du système accessibles uniquement en écriture, $\Im = \{F_1, ..., F_{|\Im|}\}$ les ressources critiques accessibles en lecture ou en écriture et enfin $\wp = \{B_1, ..., B_{|\wp|}\}$ les boîtes aux lettres utilisées par les tâches. $|R_i|$ désigne le nombre d'exemplaires de la ressource R_i et $|F_i|$ le nombre maximal de lectures simultanées de la ressource F_i.

Le corps d'une tâche est modélisé par des blocs indépendants séparés par des *Primitives Temps-Réel (PTR)*. On note $b_{i,j}$ la durée du $j^{ème}$ bloc indépendant de la tâche τ_i. Le processeur doit donc consacrer $b_{i,j}$ unités de temps au bloc pour le traiter. Le corps de chaque tâche est représenté par un ensemble de transitions mises en séquence qui représentent les blocs (voir la figure 2.3). La transition $T_{i,j,1}$ est utilisée pour modéliser un bloc d'une unité de temps, c'est à dire tel que $b_{i,j} = 1$. Les blocs de durée deux (2) ($b_{i,j} = 2$) sont modélisés par deux (2) transitions mises en séquence ($T_{i,j,1}$ *et* $T_{i,j,2}$) et enfin les blocs de durée supérieure ou égale à trois (3) unités de temps sont modélisés par trois (3) transitions mises en séquences ($T_{i,j,1}, T_{i,j,2}$ *et* $T_{i,j,3}$). La transition $T_{i,j,3}$ ne peut être tirée qu'une fois que $T_{i,j,1}$ a été tirée une fois et que $T_{i,j,2}$ a été tirée $b_{i,j} - 2$ fois.

L'exemple de la figure 2.3 modélise une tâche $\tau_i(2, 8, 9, 9)$ décomposée en 3 blocs de durées respectives 5, 2 et 1 et dont la date de première activation est 2. Elle envoie un message à son activation, prend la ressource critique R au début du second bloc et la libère à la fin du bloc. La plateforme est composée de 2 processeurs.

La première place de chaque tâche τ_i, nommée $Activ_i$ peut contenir deux couleurs de jetons : a pour tâche activée et b pour tâche qui a fini son exécution lors de sa dernière activation. La notation $a + b$ signifie a et b. Un marquage égal à $a + b$ exprime le fait qu'une instance a été activée et qu'il n'y a pas d'autres instances en cours d'exécution. Ceci permet de garantir la non réentrance. La place $Activ_i$ doit contenir un jeton b lorsque l'horloge locale contient plus de D_i jetons (ce qui exprime le fait que l'instance en cours doit avoir terminé son exécution) et ne doit jamais contenir uniquement un jeton a, ce qui signifierait qu'une instance a été activée alors que la précédente n'a pas encore ter-

$Tâche \quad \tau_i(2,8,9,9)$
$Begin$
$send(message, Mbx1);$
$Bloc_1(d=5)k;$
P(R);
Bloc$_2(d=2)$;
$V(R);$
$Bloc_3(d=1);$
$End;$

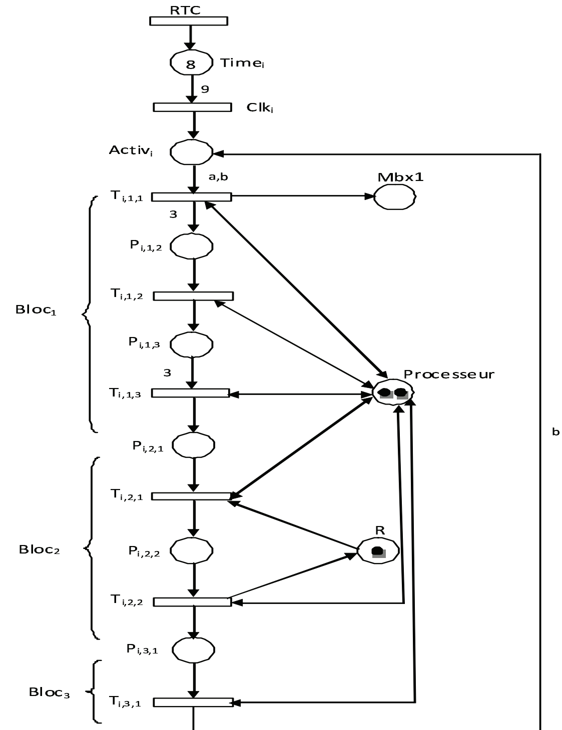


Figure 2.3 – Décomposition en bloc de la tâche τ_i

miné son exécution. Étant donné qu'on ne s'intéresse qu'aux comportements valides des tâches (les tâches respectant leurs échéances), le réseau se voit associer les contraintes terminales suivantes sur les marquages :

$$(M(Time_i) = 1 \Rightarrow M(Activ_i) = \{a,b\} \, ou \, \{b\}) \wedge$$

$$(M(Time_i) \geq D_i + 1 \Rightarrow M(Activ_i) = \{b\}) \, .$$

Ces relations définissent l'ensemble terminal Ψ. Le marquage initial des places $Activ_i$ et $Time_i$ permet de prendre en compte la date de première activation de la tâche dans l'hypothèse $r_i \leq T_i + 1$ (Le lecteur intéressé par le cas $r_i > T_i + 1$ peut se référer aux travaux de [60]). Le marquage initial est défini par :

- $M_0(Processeur) = m$ où $m \geq 1$, est le nombre de processeurs
- $\forall i = 0..n, M_0(Time_i) = \begin{cases} 0 \;\; si \;\; r_i > T_i + 1 \\ T_i - r_i + 1 \;\; sinon \end{cases}$,

- $\forall i = 0..n, M_0(Activ_i) = \begin{cases} \{a,b\} & si\ r_i = 0 \\ \{b\} & si\ r_i > 0 \end{cases}$,
- $\forall i = 1..|\Re|, M_0(R_i) = |R_i|$,
- $\forall i = 1..|\Im|, M_0(F_i) = |F_i|$,
- $\forall i = 1..|\wp|, M_0(B_i) = 0$,
- $\forall i = 0..n, j \subseteq \{1..C_i\}, k \subseteq \{1..3\}, M_0(P_{i,j,k}) = 0$.

L'ensemble des comportements valides du système ainsi modélisé est obtenu grâce au graphe des marquages accessibles. Ces comportements sont les séquences d'ordonnancement et chaque séquence d'ordonnancement est un mot du graphe des marquages. Du fait que les processeurs sont "banalisés" dans notre modèle, le problème du placement n'est pas envisagé.

Afin de déterminer l'ensemble des séquences d'ordonnancement valides de la configuration de tâches à partir du graphe des marquages terminaux accessibles, nous définissons pour le réseau un alphabet $\sum_S = \{\tau_0, ..., \tau_n\}$ où une lettre est le nom d'une tâche de la configuration. La fonction d'étiquetage associée $\sigma_S : \mathbb{T} \rightarrow \sum_S$ est définie par : $\forall T_{i,j,k} \in \mathbb{T}, \sigma_S(T_{i,j,k}) = \tau_i,\ et \sigma_S(t) = \varepsilon$ pour toute autre transition $t \in \mathbb{T}$. Un mot du langage du réseau ainsi étiqueté est une séquence d'ordonnancement du système de tâches modélisé. L'ensemble de toutes les séquences d'ordonnancement valides du système de tâches modélisé est le centre du langage terminal $L(R, \sigma, \Psi)$ du réseau autonome coloré fonctionnant sous la règle de tir maximal. Ψ est formellement défini par :

$$\Psi = \left\{ M \subseteq (\mathbb{N} \times \{a\} \times \{b\})^{|\mathbb{P}|} \right\}_{i=0...n} |$$

$$\{M(Time_i) = 1 \Rightarrow M(Active_i) = \{a,b\}\ ou\ \{b\}\} \wedge$$

$$\{M(Time_i) \geq D_i + 1 \Rightarrow M(Activ_i) = \{b\}\}$$

2.2.3 Analyse du système

L'analyse du modèle passe par la construction du graphe des marquages associé au réseau et par la construction de l'ensemble des séquences valides à partir du graphe

des marquages, avec extraction des séquences optimales par application de critères de sélection.

2.2.3.1 Profondeur du graphe des marquages accessibles

Nous avons vu dans la sous section précédente que le graphe de marquage matérialise l'évolution temporelle du système de tâches modélisé. Cette évolution temporelle devient périodique de période T à partir d'un certain moment à cause de la périodicité des tâches modélisées. Ceci découle des résultats sur la cyclicité des ordonnancements [37, 38, 43]. La cyclicité des ordonnancement a donc une incidence sur la profondeur du graphes des marquages accessibles à construire. Par conséquent, la graphe de marquage présentera une zone bouclée. Le graphe des marquages est donc fini.
Si le système de tâches considéré est à départs simultanés, le régime permanent (l'instant de début de l'évolution périodique) commence à l'instant 0. Les tâches sont toutes dans le même état aux instants 0 et T (par conséquent le marquage à la profondeur T est égal au marquage initial). Le graphe des marquages a donc une profondeur T. Chaque chemin dans le graphe des marquages a par conséquent une longueur T. Si le système n'est pas à départs simultanés, nous avons vu dans [38] que dans un contexte multiprocesseur la date d'entrée dans la phase cyclique reste un problème ouvert. Il est donc difficile de donner a priori la profondeur du graphe dans ce contexte. La construction du graphe des marquages se fera donc par simulation du réseau et en associant un critère d'arrêt. Le critère d'arrêt est inspiré de la caractérisation de phase cyclique dans les ordonnancements. En effet, l'ordonnancement du système de tâche τ est cyclique à partir d'un instant t_0 si $ST_\tau(t_0) = ST_\tau(t_0 + T)$. Cela veut dire que lorsque le système commence son régime permanent à partir de t_0 alors,

$$\forall \tau_i \in \tau, \forall t > t_0 + T, RCT_i(t) = RCT_i(t-T) \wedge dist_i(t) = dist_i(t-T).$$

Sur le graphe des marquages, chaque marquage inséré à la hauteur h est comparé aux marquages se trouvant T niveaux plus haut dans le graphe. Si $M(h) = M(h-T)$, alors Le marquage $M(h)$ est une feuille. Cela veut dire qu'on a atteint le marquage correspondant

à la zone bouclée. Dans la suite nous notons L la profondeur du graphe. $L = T$ pour un système de tâches à départs simultanés car le régime permanent commence à l'instant 0.

2.2.3.2 Taille du graphe des marquages accessibles

Comme dans tout système modélisé par un réseau de Petri, nous sommes confrontés au problème d'*explosion combinatoire* : la taille du graphe peut évoluer exponentiellement avec la dimension du réseau. En effet, soit τ une configuration de n tâches. Sur la durée de simulation L, chaque tâche τ_i a $\left\lceil \frac{L-r_i}{T_i} \right\rceil$ instances. Au plus $\left\lceil \frac{L-r_i}{T_i} \right\rceil C_i$ transitions de la tâche τ_i vont être tirées pendant la durée L de la simulation. Du fait que pour les systèmes à départs différés il y a une phase de montée en charge suivi de la phase cyclique, le graphe des marquages est décomposé en deux parties. Une première partie représente la montée en charge du système et la seconde partie correspond à la phase cyclique. La première partie du graphe est contenue dans un hyperpavé de profondeur t_c où t_c est la date d'entrée dans la phase cyclique et la seconde partie dans un hyperpavé de profondeur T. Dans les deux cas l'hyperpavé est de dimension n (les n tâches du système et la tâche oisive). Les cotés de chaque hyperpavé sont de taille respectif $\left\lceil \frac{L-r_1}{T_1} \right\rceil C_1, \left\lceil \frac{L-r_2}{T_2} \right\rceil C_2, \ldots, \left\lceil \frac{L-r_n}{T_n} \right\rceil C_n$. Le nombre de sommets de chaque hyperpavé est borné par $\prod_{i=1}^{i=n} \left(\left\lceil \frac{L-r_i}{T_i} \right\rceil c_i + 1 \right)$. Cette borne est surestimée car [60] a montré que les contraintes temporelles et structurelles (partage de ressources et communication et entre tâches) réduisent la taille théorique de l'hyperpavé.

2.2.3.3 Construction du graphe des marquages accessibles

L'évolution temporelle du système de tâches modélisé est matérialisée par l'évolution temporelle du réseau de Petri étiqueté le représentant. L'évolution temporelle du réseau est donc décrite par le graphe des marquages étiqueté par l'alphabet

$$\sum_S = \{\tau_1, \ldots, \tau_n\} = \{\sigma_S(t_i) | t_i \in \mathbb{T} \wedge \sigma_S(t_i) \neq \varepsilon\}.$$

Ainsi donc, le tir d'un ensemble de transitions correspond à l'ordonnancement des tâches associées à l'ensemble.

Rappelons que le réseau de Petri fonctionne sous la règle de tir maximal et que la place processeur contient m jetons. Dans ce contexte donc, on considère des *tirs groupés*. *Un groupe* correspond aux *tirs simultanés* de plusieurs transitions (en plus des transitions du système de tâches on peut tirer des transition du système d'horloge) et la même transition ne peut être tirée plusieurs fois. On représente donc un *groupe* par un ensemble de transitions noté $\omega_h = n_1\tau_1 + \ldots + n_n\tau_n$ avec $n_i \in \{0,1\}, i = 1,\ldots,n$ (un tir d'une transition du système d'horloge donne le mot vide).
Il n'est pas efficace de construire le graphe des marquages accessibles puis ensuite de ne garder que les marquages de Ψ du fait du problème d'explosion combinatoire décrit plus haut. Pour maîtriser cette explosion, nous combinons l'approche originale proposée par Emmanuel Grolleau dans [60] qui permet de réduire les retour en arrière et le critère d'arrêt que nous avons proposé. Pour rappel la construction du graphe s'arrête dès que la séquence cyclique est détectée.
Pour réduire les retours en arrière, à chaque profondeur h du graphe, et pour toute tâche τ_i, $RCT_i(h)$ est sauvegardée car la connaissance de $RCT_i(h)$ pour toute tâche permet de prévoir les dépassements d'échéance. En effet, le marquage à la profondeur h décrit l'état du système à l'instant $t = h$. La connaissance de $RCI_i(t)$ qui est la charge restante à traiter pour terminer la tâche τ_i permet de prédire si la tâche peut encore espérer respecter son échéance ou pas. La laxité dynamique de la tâche τ_i à l'instant t est donnée par $L_i(t) = k.T_i + r_i + D_i - t$. Une condition suffisante pour que chaque tâche puisse respecter son échéance est alors $RCI_i(t) \leq L_i(t), \forall \tau_i \in \tau$. La connaissance de $RCT_i(h)$ pour toute tâche permet donc d'éviter le tir d'une transition qui conduira à une violation d'échéance dans le futur.
Dans le cas monoprocesseur, Emmanuel Grolleau dans [60] a ajouté la *contrainte de successeur*, une contrainte qui permet de diminuer les changements de contexte inutiles, pour réduire la taille du graphe. Les préemptions ne peuvent avoir lieu que si une tâche se reactive, une tâche termine, une ressource est demandée, une ressource libérée, un

message est émis, un message est attendu.... Le principe de cette contrainte est de limiter l'entrelacement de blocks indépendants. Cette contrainte ne peut être utilisée dans le cas multiprocesseur car elle peut conduire à des dépassements d'échéances comme l'illustre la figure 2.4. Néanmoins nous pourront diminuer les changements de contexte en minimisant le nombre de préemptions lors du placement des tâches.

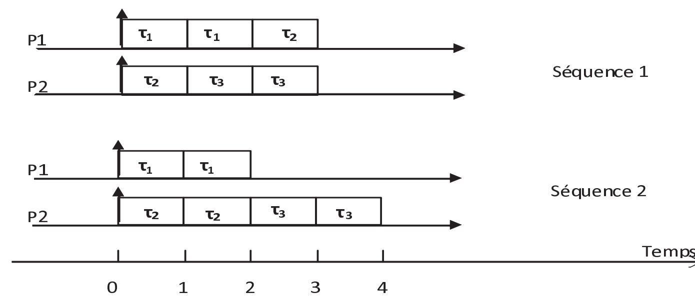


Figure 2.4 – Contrainte de successeur
Deux séquences d'ordonnancement du système de tâche $\tau = \{\tau_1 = (0,2,3,3), \tau_2 = (0,2,3,3), \tau_3 = (0,2,3,3)\}$. *Sur la deuxième séquence on interdit la préemption de la tâche* τ_2*, ce qui entraîne un dépassement d'échéance de la tâche* τ_3

2.2.3.4 Le placement des tâches

Une séquence obtenue par l'approche précédente doit ensuite être mise en œuvre effectivement. Pour cela, il faut pour finir réaliser l'assignation des tâches aux processeurs. Nous introduisons pour cela la notion de placement qui est l'étape qui consiste en l'affectation des tâches aux processeurs. Le placement est une fonction $A : \tau \times \mathbb{N} \longrightarrow P$ (pour les tâches non ordonnancées à l'instant t) telle que $A(\tau_i, t) = p_j$ signifie que la tâche τ_i a été placée à l'instant t sur le processeur p_j.
Connaissant la séquence d'ordonnancement, il reste à placer les tâches sur les processeurs tout en minimisant les changements de contexte pour réduire le surcoût processeur engendré par les changements de contexte. L'algorithme 1 que nous proposons généralise l'heuristique proposée dans [10]. Cette heuristique minimise les changements de contexte dans le cadre des ordonnancements P-équitables.

Algorithme 1. ***Entrée*** *: Un ordonnancement valide S_1 de longueur L*

Sortie *: Une affectation A des tâches aux processeurs*

Soit $S_1(1) = \{\tau_{i_1}, ..., \tau_{i_{m'}}\}$ *avec* $m' \leq m$

$A(\tau_{i_j}, 1) \leftarrow p_j$ *// à t = 1, l'affectation est arbitraire.*

Pour $t = 2$ ***à*** *L*

Si $S_1(t) \cap S_1(t-1) = \emptyset$ ***Alors***

Soit $S_1(t) = \{\tau_{i_1}, ..., \tau_{i_{m'}}\}$ *avec* $m' \leq m$

$A(\tau_{i_j}, t) \leftarrow p_j$ *// Ici aussi l'affectation est arbitraire.*

Sinon

$\forall \tau_{i_j} \in S_1(t) \cap S_1(t-1)$, $A(\tau_{i_j}, t) \leftarrow A(\tau_{i_j}, t-1)$.

// τ_j *doit être affectée au même processeur qu'à l'instant* $t-1$.

$\forall \tau_{i_j} \notin S_1(t) \cap S_1(t-1)$, $A(\tau_{i_j}, t) \leftarrow p_k$ *où k est le plus petit entier inférieur ou égal à m tel que* p_k *n'est pas encore alloué*

fin Si

fin Pour

2.2.3.5 Mise oeuvre dans PeNSMARTS

Le logiciel PeNSMARTS pour Petri Nets Scheduling, Modeling and Analysis of Real-Time Systems est la résultante du travail théorique réalisé dans [59] et [60]. C'est un outils de modélisation et de validation d'application temps-réel à base de réseau de Petri. Il permet, à partir d'une application modélisée par un réseau de Petri autonome coloré et fonctionnant sous la règle de tir maximal, la création de séquences d'ordonnancement hors-ligne. Les critères d'optimisation intégrés dans le noyau permet de sélectionner des séquences répondant à certains critères. Le lecteur intéressé peut se référer à la thèse d'Emmanuel Grolleau [60] pour plus de détails sur PeNSMARTS.

Dans sa forme actuelle, PeNSMARTS ne prend en compte que des applications fonctionnant sur une plateforme monoprocesseur. Les études à venir ont pour objectif d'étendre les fonctionnalités de PeNSMARTS aux plateformes multiprocesseurs.

2.3 Conclusion

Dans ce chapitre nous avons présenté les ordonnancements hors-lignes à base de modèle. Nous avons particulièrement insisté sur les ordonnancements à base de réseaux de Petri. Ce chapitre s'appuie essentiellement sur les travaux de thèse d'Emmanuel Grolleau [60].

Deuxième partie

Contribution

Dans cette partie, nous présentons notre contribution à l'analyse de l'ordonnancement des applications temps-réel sur une plateforme multiprocesseur. La panne d'un processeur est un évènement possible dans la vie d'un système multiprocesseur et cela impose de disposer de méthodes performantes d'analyse de l'impact d'une panne sur l'ensemble du système pour la maîtrise du surcoût engendré par la politique de tolérance à la panne processeur.

Un autre facteur d'évolution du système consiste en un changement dans les besoins. Cette évolution est généralement de nature fonctionnelle. Il s'agit d'étendre la capacité du système pour s'adapter à un environnement modifié ou pour favoriser des interactions avec d'autres systèmes. Dans le premier cas, les fonctions existantes doivent être modifiées ou s'enrichir de nouvelles fonctions. Dans le second cas, il s'agit de créer une nouvelle interface avec un autre système. Dans tous les cas, l'évolution se caractérise par l'adjonction de nouvelles tâches dans le système ou la suppression de tâches du système. Nous ne parlons pas de modification d'une tâche (cela peut concerner ses paramètres ou son corps même) car modifier une tâche peut revenir à supprimer la tâche et à la remplacer par la même tâche avec les modifications associées.

Notre contribution a pour objectif de :

- prévoir les reconfigurations en cas de panne processeur
- prévoir l'adjonction de nouvelles tâches qui peuvent être apériodiques ou périodiques.

Pour illustrer les stratégies proposées et les évaluer, des simulations ont été conduites. Pour cela un générateur aléatoire de systèmes a été développé ainsi qu'un outil de simulation et de visualisation d'ordonnancements temps-réel.

I- Le générateur aléatoire d'applications temps-réel

Une application temps-réel est une application multitâches et chaque tâche de l'application est appelée tâche temps-réel. Nous avons vu qu'une tâche temps réel τ_i se caractérise par ses quatre paramètres temporels (r_i, C_i, D_i, T_i). La génération d'une tâche consiste donc en la génération de ses paramètres temporels. Les paramètres temporels

d'une tâche τ_i sont modélisés par de variables aléatoires.
Les périodes sont tirées selon une loi uniforme de sorte à limiter l'hyperpériode des systèmes de tâches. Nous avons utilisé pour cela l'approche proposé par Macq et Goossens dans [88]. Ils proposent une technique de génération de nombres entiers de sorte que leur *ppcm* soit borné.
Le délai, la pire durée d'exécution et la date de première activation d'une tâche τ_i sont tirés aléatoirement suivant une loi uniforme respectivement dans les intervalles $[0,T_i]$, $[1,D_i-1]$ et $[0,T_i]$.

II- Le simulateur d'ordonnancements temps-réel

Le simulateur d'ordonnancement est un outil qui prend en entrée une configuration de tâches et nous fournit une séquence d'ordonnancement suivant l'un des algorithmes d'ordonnancement temps réel. Les algorithmes d'ordonnancement intégrés sont les algorithmes *RM*, *EDF*, *LLF* et *PF* et PD^2.
Nous considérons des plateformes multiprocesseurs, ce qui pose le problème de la taille de séquence à produire puisque le problème de la cyclicité reste ouvert dans ce contexte. Pour les systèmes à départs simultanés, la taille de la séquence est le *PPCM* des périodes des tâches de l'application. Ceci pose un problème d'ordre pratique. En effet, pour pouvoir visualiser la séquence produite, il faut que la taille de la séquence à produire soit "raisonnable". Pour cela, il faut borner le *PPCM* des périodes des tâches à générer. Cela justifie l'approche utilisée pour la génération des périodes des tâches.
Nous avons intégré d'autres fonctionnalités dans l'outil de simulation d'ordonnancement. Elles concernent notamment la prise en compte d'une panne processeur et de l'arrivée d'un flux de tâches apériodiques. Les détails d'implémentation de cet environnement intégré de simulation sont présentés dans les chapitres à venir.
Dans cette partie, nous nous intéressons tout d'abord au problème de la cyclicité dans le contexte des ordonnancements sur une architecture multiprocesseur. Ceci constitue le chapitre 4. Dans le chapitre 5, nous étudions le problème de la reconfiguration en cas de panne processeur. Nous présentons dans le chapitre 6 les extensions que nous avons apportées dans le cadre de la mise en œuvre des ordonnancements P-équitables et nous

terminons par le chapitre 7 où nous concluons et proposons nos perspectives. Dans nos perspectives, nous présenterons le travail en cours et ce qui reste à faire.

CHAPITRE 3

ETUDE DU PROBLÈME DE LA CYCLICITÉ DES ORDONNANCEMENTS MULTIPROCESSEURS

Comme nous l'avons déjà évoqué, qu'il s'agisse de procéder à des analyses basées sur des simulations, ou de procéder à une analyse d'ordonnançabilité à base de modèle, nous devons répondre aux deux questions fondamentales suivantes :

- quelle est la taille du plan d'exécution ?
- quand commence la phase périodique du plan d'exécution ?

Plusieurs travaux dans la littérature se sont intéressés à ces questions. Il a été établi dans les travaux de [42, 78] que la période du plan d'excécution est égale à l'hyperpériode des tâches. Le problème qui se pose est alors la connaissance du début de la phase périodique dans le plan. Ce problème est connu sous le nom de *problème de la cyclicité*. Une solution au problème de cyclicité permettrait donc répondre aux deux questions fondamentales et donc de limiter l'espace d'exploration si une approche hors-ligne basée sur des modèles est utilisée. Le problème de la cyclicité a trouvé une solution dans le cadre des ordonnancements monoprocesseurs dans les travaux de [37, 78]. Les travaux de [78] donnent une borne à la taille du plan d'exécution dans le cadre des ordonnancements à priorités fixes et pour des tâches indépendantes, et les travaux de [37] améliorent cette borne et la généralise aux algorithmes d'ordonnancement classiques, et pour des systèmes de tâches interagissantes. Ils établissent que la date d'entrée dans la phase cyclique du plan d'exécution est indépendante de la stratégie d'ordonnancement utilisée. Malheureusement, la connaissance de la taille de ce plan d'exécution ou de la longueur de la séquence à générer est plus complexe quand on passe au contexte des ordonnancements multiprocesseurs.

3.1 Le problème de la cyclicité dans un contexte d'ordonnancement multiprocesseur

Dans le contexte des ordonnancements multiprocesseurs, très peu de travaux existent concernant la cyclicité. Les résultats établis dans les travaux de [37] concernant les ordonnancements monoprocesseurs ne tiennent plus quand on passe en multiprocesseur. Les premiers travaux qui se sont intéressés au problème de la cyclicité dans ce contexte furent les travaux réalisés par Annie Geniet dans [33], Liliana Cucu et Joël Goossens dans [43], et enfin Annie Geniet et Sadouanouan Malo dans [38]. Les travaux de [33] ont caractérisé l'entrée dans la phase cyclique pour les ordonnancements à priorités fixes pour des systèmes de tâches indépendantes, et sur une architecture composée de deux processeurs. Ces travaux ont ensuite été généralisés dans [38] à une architecture composée d'un nombre quelconque m de processeurs et à une famille d'algorithmes d'ordonnancement qui contient tous les algorithmes classiques. Il s'agit de la classe des ordonnancements monotones. Une conjecture a ensuite été établie quand les ordonnancements à priorités dynamiques sont considérés. Enfin, dans leur travaux [43], L. Cucu et J. Goossens ont donné une méthode analytique pour le calcul de la date d'entrée dans la phase cyclique. Ils ont montré que dans le cadre des ordonnancements déterministes à priorités fixe pour des systèmes de tâches indépendantes, le régime permanent commence à l'instant S_n défini de manière inductive par :

$$\begin{cases} S_1 = r_1 \\ S_i = max\left\{r_i, r_i + \left\lceil \frac{S_{i-1} - r_i}{T_i} \right\rceil .T_i\right\} \forall i \in \{2, 3, \ldots n\} \end{cases}$$

La principale difficulté liée à l'étude de la cyclicité dans le contexte multiprocesseur est due à l'interdiction du parallélisme d'exécution des tâches pouvant entrainer une inactivité processeur malgré une charge en attente non nulle. Des expérimentations en multiprocesseur nous ont aussi montré dans [38] que la date d'entrée dans le régime permanent dépend de la stratégie d'ordonnancement utilisée.

Avant de présenter notre contribution sur l'étude de la cyclicité en multiprocesseur, nous allons d'abord introduire les notions de temps creux cycliques et acycliques et montrer

que dans le contexte des ordonnancements multiprocesseurs, le nombre de temps creux acycliques demeure borné. Ce résultat étend les résultat de [37] aux architectures multiprocesseurs.

3.2 Notion de temps creux cycliques et acycliques

Soit $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ un système de n tâches s'exécutant sur une plateforme composée de m processeurs identiques. Notons U le facteur d'utilisation total des processeurs par le système de tâche. Si $U < m$, alors le système n'est pas de charge maximale et reste inactif pendant $T(m-U)$ unités de temps toutes les hyperpériodes. Ces temps d'inactivité des processeurs sont appelés *temps creux cycliques*. Ils ont lieu toutes les hyperpériodes

Si la charge du système est égale à m, il n'y a pas de temps creux cycliques. Pourtant, on constate parfois la présence de temps creux résiduels. Ces temps creux, qui ne sont pas périodiques, sont dits *temps creux acyliques*. Ils s'observent pendant la montée en charge du système qui correspond à une période d'instabilité du système.
La figure 3.1 illustre la présence de ces temps creux. Cette figure présente une séquence d'ordonnancement de la configuration de tâches

$$\tau = \{\tau_1(0,1,3,3), \tau_2(0,1,3,3), \tau_3(0,4,9,9), \tau_5(8,2,9,9)\}.$$

Les tâches sont ordonnées par ordre décroissant des priorités fixes (la tâche τ_1 a la priorité la plus élevée et τ_5 la priorité la plus faible). Le système de tâches est ordonnancé sur 2 processeurs et son facteur d'utilisation total est égal à 2. La charge étant maximale, il n'y a pas de temps creux cycliques. Néanmoins on constate la présence d'un temps creux à l'instant 7. Par ailleurs, la séquence se répète à partir de l'instant 8 et ne comporte plus de temps creux. Le temps creux observé à l'instant 7 est un temps creux acyclique.

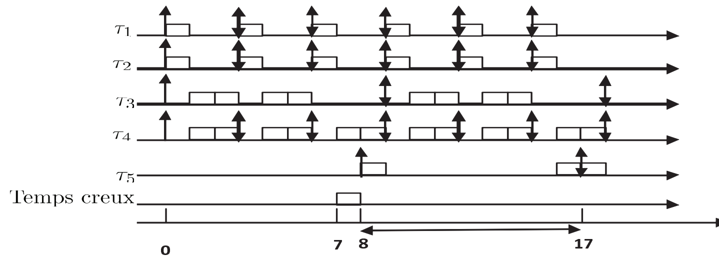


Figure 3.1 – Temps creux acyclique :
cette figure montre la séquence d'ordonnancement du système de tâche $\tau = \{\tau_1(0,1,3,3), \tau_2(0,1,3,3), \tau_3(0,4,9,9), \tau_5(8,2,9,9)\}$. *Un temps creux acyclique est observé à l'instant 7 sur un processeur.*

Dans leurs travaux [37], Emmanuel Grolleau et Annie Geniet ont montré que le nombre de temps creux acyclique est borné et indépendant de la stratégie d'ordonnancement choisie dans un contexte d'ordonnancement monoprocesseur et la séquence devient cyclique après l'apparition du dernier temps creux acyclique si l'ordonnancement est conservatif et déterministe. De plus, si t_c désigne la date d'apparition du dernier temps creux acyclique, ils ont établi que t_c est indépendant de la stratégie choisie et que

$$t_c \leq max\{r_i\}_{\tau_i \in \tau} + T.$$

Quand on considère le cas multiprocesseur, ces résultats ne tiennent plus. En effet, la figure 3.2 donne un autre ordonnancement possible de la configuration de tâches τ. L'algorithme utilisé n'est pas à priorités fixes. Cette fois, il y a deux temps creux acycliques aux instants 2 et 5 et l'entrée dans la phase cyclique se fait à l'instant 8 alors que le dernier temps creux acyclique apparaît à l'instant 5. Le régime permanent ne peut pas commencer à l'instant 6 car à cet instant la tâche τ_5 n'est pas encore active et entre les instants 6 et 15 elle n'a pas été ordonnancée. On ne peut donc pas généraliser le résultat obtenu dans [37] au cas multiprocesseur. Des travaux supplémentaires sont donc nécessaires pour une caractérisation du début du régime permanent.

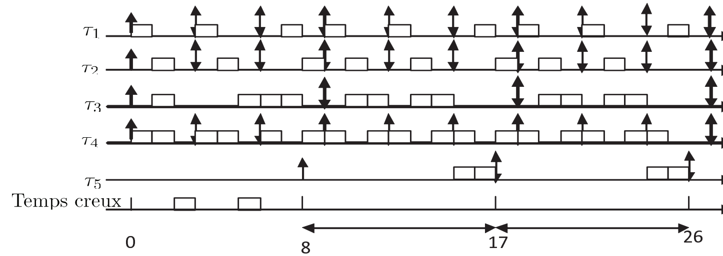


Figure 3.2 – Début du régime permanent :
cette figure montre un autre ordonnancement possible de la configuration de tâches $\tau = \{\tau_1(0,1,3,3), \tau_2(0,1,3,3), \tau_3(0,4,9,9), \tau_5(8,2,9,9)\}$. *On observe deux temps creux et le dernier temps creux s'observe à l'instant 5. Le régime permanent commence cependant à l'instant 8.*

Rappelons que pour un système de charge $U < m$, il existe dans ce cas $T(m-U)$ temps creux cycliques dans le système toutes les hyperpériodes. Posons $k = T(m-U)$. Nous introduisons alors k nouvelles tâches $\tau_{n+1}, \tau_{n+2}, \ldots, \tau_{n+k}$ pour modéliser les temps creux cycliques. Ces tâches sont appelées tâches oisives et ont les paramètres temporels suivants : $r_{n+i} = 0, C_{n+i} = 1, D_{n+i} = T_{n+i} = T$. Ces tâches comblent les temps d'inactivité processeur et on se ramène à un système de charge maximale. Les résulats obtenus dans le cadre des système à charge maximale peuvent donc s'appliquer dans ce cas.
De plus, le fait d'introduire les tâches oisives permet alors de prendre en compte les ordonnancements non conservatifs puisque les tâches oisives peuvent être par moment ordonnancées avant certaines tâches du système. On parle alors de *pseudo-conservatisme*.
Notons que, contrairement au cas monoprocesseur, on ne peut pas introduire une unique tâche oisive de durée $T(m-U)$, et ce pour deux raisons :

1. il se peut que $T(m-U)$ soit supérieur à T, or nous ne considérons que des tâches dont le facteur d'utilisation est inférieur à 1, puisque nous refusons la parallélisation ;

2. la tâche oisive devrait pouvoir s'exécuter en parallèle, car plusieurs processeurs peuvent être inactifs simultanément. Une seule tâche oisive ne rentre donc pas dans le cadre de notre modélisation.

Avant de caractériser le début du régime permanent, nous commençons par établir que le nombre de temps creux acyclique reste borné quand on passe au contexte multi-processeur à travers la proposition suivante suivant :

Proposition 1. *Soit S un système de tâches et O un ordonnancement valide. Alors le nombre de temps creux acycliques dans O est borné.*

Preuve :
Grâce au principe de complétion, nous pouvons nous ramener, sans perte de généralité, au cas d'un système de charge maximale ($U = m$). Considérons l'intervalle $[0, r + kT]$ avec $k \in N$ et $r = max\{r_i\}_{i=1...n}$. Soit τ_i une tâche. Le nombre d'instances complètes de la tâche τ_i dans cet intervalle est égal à $\left\lfloor \frac{r-ri}{T_i} \right\rfloor + \frac{kT}{T_i}$. Pour son exécution, la tâche τ_i aura donc besoin de $\left(\left\lfloor \frac{r-ri}{T_i} \right\rfloor + \frac{kT}{T_i} \right) C_i$ unités de temps processeur. Il s'en suit alors que les processeurs sont actifs globalement pendant au moins

$$\left(\sum_{\tau_i \in \tau} \left(\left\lfloor \frac{r-ri}{T_i} \right\rfloor + \frac{kT}{T_i} \right) C_i \right) = \left(\sum_{\tau_i \in \tau} \left\lfloor \frac{r-r_i}{T_i} \right\rfloor C_i \right) + mkT$$

unités de temps (nous rappelons que $U = m$).
Par conséquent, sur cet intervalle d'étude, le système reste inactif pendant au plus

$$m(r+kT) - \left[\left(\sum_{\tau_i \in \tau} \left\lfloor \frac{r-r_i}{T_i} \right\rfloor \right) C_i + mkT \right] = mr - \left(\sum_{\tau_i \in \tau} \left\lfloor \frac{r-r_i}{T_i} \right\rfloor \right) C_i$$

unités de temps.
Ce majorant du nombre de temps creux ne dépend pas de k. Donc le nombre de temps creux reste borné quelle que soit la taille de l'intervalle d'étude. Il s'en suit que le nombre de temps creux acycliques est borné. □

Dans les sections qui suivantes, nous présentons notre contribution à l'analyse de la cyclicité des ordonnancements temps-réel multiprocesseurs. Nous caractérisons le

début du régime permanent dans un cadre plus général des ordonnancements pseudo-conservatifs déterministes et *monotones*. Nous établissons ensuite que les ordonnancements à *priorités fixes*, les ordonnancements à *priorités dynamiques* comme *EDF* et *LLF* et enfin les ordonnancements *P-équitables* comme *PF* et PD^2 font partie de la *classe des ordonnancements monotone*.

3.3 Etude de la cyclicité pour la classe des ordonnancements monotones

Les ordonnancements monotones sont tels que, pour toute tâche, la charge exécutée par l'instance en cours est toujours supérieure ou égale à la charge exécutée par l'instance en cours une hyperpériode plus tard.

Pour rappel, *Un ordonnancement déterministe conservatif O est monotone si*

$$\forall \tau_i \in \tau,\ \forall t \geq r_i,\ RCT_i(t+T) \geq RCT_i(t).$$

Cela peut aussi être exprimé de la façon suivante :

$$\forall \tau_i \in \tau,\ \forall t \geq r_i,\ RCT_i(t) + \overline{RCT}_i(t+T) \leq C_i.$$

Dans cette partie, nous montrons que pour un ordonnancements monotone, pseudo-conservatif et déterministe, le régime permanent commence après le dernier temps creux acyclique. L'existence de ce dernier temps creux acyclique découle de la proposition 1 qui montre que le nombre de temps creux acycliques est borné.

Theorème 4. *Dans un ordonnancement déterministe, (pseudo)conservatif et monotone O, pour un système de tâches indépendantes, si un temps creux est obtenu à l'instant t_c, suivi de T unités de temps sans temps creux, alors l'ordonnancement obtenu dans la fenêtre temporelle $[t_c+1, t_c+T+1]$ correspond au régime permanent de l'application.*

Formellement, cela s'énonce de la façon suivante :

$$\begin{cases} |O(t_c)| < m \\ \forall t \in [t_c+1, t_c+T+1], |O(t)| = m \end{cases} \Rightarrow O \text{ est cyclique à partir de } t_c+1$$

La suite de cette section est consacrée à la preuve du théorème. La preuve est basée sur l'analyse de la charge processeur. Sauf mention explicite les ordonnancements que nous considérons sont déterministes et (pseudo)conservatifs. L'idée de la preuve est de montrer que $ST_S(t_c+1) = ST_S(t_c+T+1)$ pour pouvoir appliquer le lemme 1.

3.3.1 Temps creux complet - temps creux partiel

Lors d'un temps creux, il peut arriver que plusieurs ou la totalité des processeurs soient inactifs au même instant amenant donc à distinguer deux types d'inactivité processeur (voir les figure 3.3 et 3.4) :

- l'inactivité simultanée de tous les processeurs que nous appellerons *temps creux complet* ;
- l'inactivité de plusieurs processeurs mais pas la totalité appelée *temps creux partiel* : q processeurs ($0 < q < m$) sont inactifs.

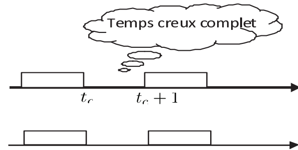


Figure 3.3 – Temps creux complet sur 2 processeurs

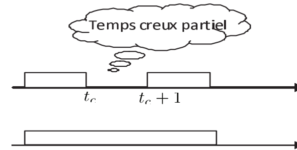


Figure 3.4 – Temps creux partiel sur 2 processeurs

Pour la preuve du théorème, nous supposons que le facteur d'utilisation total de l'application est égal à m. Ceci peut se faire sans perte de généralité selon le principe de complétion. Il n'y a donc pas de temps creux cyclique dans l'ordonnancement. Nous étudierons le cas où le dernier temps creux acyclique à l'instant t_c est complet puis le cas où il est partiel. Si aucun temps creux n'apparait, alors t_c prendra la valeur -1.

3.3.2 Cas où le dernier temps creux acyclique est complet

Nous supposons que le temps creux à l'instant t_c est un temps creux complet. Nous partitionnons l'ensemble des tâches en trois sous ensembles :

- RE correspond aux tâches pour lesquelles $t_c + 1$ est une date d'activation (éventuellement la date de première activation) :

$$\tau_i \in RE \Leftrightarrow (t_c + 1 - r_i) \ mod \ T_i = 0 \ et \ r_i \leq t_c + 1.$$

- LR correspond aux tâches ayant un réveil tardif, c'est à dire dont la première activation a lieu après $t_c + 1$:

$$\tau_i \in LR \Leftrightarrow r_i > t_c + 1.$$

- AR correspond aux tâches actives telles que $r_i \leq t_c$ et $t_c + 1$ n'est pas un instant de réactivation :

$$\tau_i \in AR \Leftrightarrow r_i \leq t_c \ et \ (t_c + 1 - r_i) \ mod \ T_i \neq 0.$$

La preuve du théorème est basée sur l'analyse de la charge processeur entre $t_c + 1$ et $t_c + T + 1$. Le système étant supposé de charge maximale, et puisqu'aucun temps creux n'intervient entre $t_c + 1$ et $t_c + T + 1$ la charge $W(t_c + 1, t_c + T + 1)$ vaut mT.

3.3.2.1 Calcul de la charge processeur entre $t_c + 1$ et $t_c + T + 1$

Le système de tâches étant partitionné en trois sous ensembles, nous allons calculer la contribution de chacun d'entre eux à la charge processeur totale.

- Contribution des tâches du sous ensemble RE : chaque tâche τ_i dans RE s'exécute exactement $\frac{T}{T_i}$ fois dans cette fenêtre temporelle. La charge processeur engendrée par l'ensemble des tâches de RE est donc égale à : $\sum_{\tau_i \in RE} \frac{T}{T_i} C_i$.
- Contribution des tâches du sous ensemble LR : chaque tâche τ_i dans LR termine $\left\lfloor \frac{t_c + T + 1 - r_i}{T_i} \right\rfloor$ instances dans la fenêtre temporelle et en commence une autre. La charge processeur engendrée par l'ensemble des tâches de LR est donc égale à :

$\sum_{\tau_i \in LR}\left(\left\lfloor \frac{t_c+T+1-r_i}{T_i} \right\rfloor C_i + \overline{RCT_i}(t_c+T+1)\right)$.

– Contribution des tâches du sous ensemble AR : chaque tâche τ_i de AR a terminé $\frac{T}{T_i}-1$ instances dans la fenêtre temporelle et en a commencé une autre. A l'instant t_c+1, il n' y a aucune tâche en attente puisque il y a un temps creux complet et que l'ordonnancement est conservatif. De plus t_c+1 n'est une date d'activation pour aucune des tâches de AR. La contribution à la charge processeur des tâches de AR est donc égale à :
$\sum_{\tau_i \in AR}\left(\left(\frac{T}{T_i}-1\right) C_i + \overline{RCT}_i(t_c+T+1)\right)$.

Il s'en suit que la charge processeur totale dans la fenêtre temporelle $[t_c+1, t_c+T+1]$ vaut :

$W(t_c+1, t_c+T+1) = \sum_{\tau_i \in RE} \frac{T}{T_i} C_i$
$+\sum_{\tau_i \in LR}\left(\left\lfloor \frac{t_c+T+1-r_i}{T_i} \right\rfloor C_i + \overline{RCT_i}(t_c+T+1)\right)$
$+\sum_{\tau_i \in AR}\left(\left(\frac{T}{T_i}-1\right) C_i + \overline{RCT_i}(t_c+T+1)\right)$
$= mT$.

3.3.2.2 Étude des tâches dont la date d'activation est tardive

Afin d'évaluer $\left\lfloor \frac{t_c+T+1-r_i}{T_i} \right\rfloor$ pour les tâches tardives, nous montrons qu'elles sont toutes activées moins d'une période après t_c+1. Dans le cas contraire, des temps creux apparaissent au delà de t_c, ce qui est contraire à notre hypothèse.

Lemme 2. *$\forall \tau_i \in LR$, nous avons $t_c+1 < r_i < t_c+T_i+1$.*

Preuve :

a- Supposons qu'il existe i_0 tel que $r_{i_0} > t_c+T_i+1$. Alors, nous avons $\left\lfloor \frac{t_c+T+1-r_{i_0}}{T_{i_0}} \right\rfloor < \frac{T}{T_{i_0}} - 1$. Par ailleurs, nous avons $\overline{RCT}_i(t_c+T+1) \leq C_i$ pour toute tâche et $\left\lfloor \frac{t_c+T+1-r_i}{T_i} \right\rfloor \leq \frac{T}{T_i} - 1$ pour toute tâche τ_i de LR autre que τ_{i_0}. Nous avons donc :
$mT = W(t_c+1, t_c+T+1)$
$\leq \sum_{\tau_i \in RE \cup AR \cup (LR - \{\tau_{i_0}\})} \frac{T}{T_i} C_i + \left\lfloor \frac{t_c+T+1-r_{i_0}}{T_{i_0}} \right\rfloor C_{i_0} + \overline{RCT}_{i_0}(t_c+T+1)$
$< \sum_{i=1}^{i=n} \frac{T}{T_i} C_i - C_{i_0} + C_{i_0}$
$< mT$,
ce qui est absurde.

b- Supposons maintenant qu'il existe i_0 tel que $r_{i_0} = t_c + T_{i_0} + 1$. Dans ce cas nous aurons $\left\lfloor \frac{t_c+T+1-r_{i_0}}{T_i} \right\rfloor = \frac{T}{T_{i_0}} - 1$ et $\overline{RCT_{i_0}}(t_c + T + 1) = 0$. Il s'en suit alors la même contradiction que dans le cas précédent. D'où le résultat.□

3.3.2.3 Preuve du théorème

Nous pouvons maintenant donner la preuve du théorème puisque nous avons une estimation plus fine de la contribution de chaque sous ensemble de tâches à la charge processeur globale. D'après le lemme 2, pour toute tâche τ_i de LR, $\left\lfloor \frac{t_c+T+1-r_i}{T_i} \right\rfloor = \frac{T}{T_i} - 1$.
Il s'en suit que : $mT = W(t_c + 1, t_c + T + 1)$
$= \sum_{i=1}^{i=n} \frac{T}{T_i} C_i + \sum_{\tau_i \in LR \cup AR} (\overline{RCT_i}(t_c + T + 1) - C_i)$
$= mT + \sum_{\tau_i \in LR \cup AR} (\overline{RCT_i}(t_c + T + 1) - C_i)$.

Or nous savons que $\overline{RCT}_i(t_c + T + 1) - C_i \leq 0$ car le temps processeur reçu par une instance ne peut pas être supérieure d'où $\overline{RCT}_i(t_c + T + 1) = C_i$ pour toute tâche appartenant à $LR \cup AR$.
Il s'ensuit donc que $RCT_i(t_c + 1) = RCT_i(t_c + T + 1) = 0 \; \forall \tau_i \in LR \cup AR$ et par définition de RE, $\tau_i \in RE \Rightarrow RCT_i(t_c + 1) = RCT_i(t_c + T + 1) = C_i$.
Enfin, pour toute tâche τ_i, nous avons $dist_i(t_c + T + 1) = dist_i(t_c + 1)$ puisque $dist_i$ est périodique de périodicité T_i. Alors, pour toute tâche τ_i, nous avons $st_i(t_c + T + 1) = st_i(t_c + 1)$. D'après le lemme 1, nous pouvons conclure que l'ordonnancement O est cyclique à partir de $t_c + 1$ et de périodicité T.□

3.3.3 Cas où le dernier temps creux acyclique est partiel

Supposons maintenant qu'à l'instant t_c, q tâches sont en cours d'exécution avec $0 < q < m$. Sans perte de généralité, nous supposons que les tâches $\tau_1, \tau_2, \ldots, \tau_q$ ont été ordonnancées à l'instant t_c. Nous décomposons le systèmes de tâches en quatre sous ensembles. Aux trois sous ensembles précédents, nous ajoutons un quatrième sous ensemble. Ce sous ensemble est noté PT et correspond aux tâches en cours d'exécution à l'instant t_c ($PT = \{\tau_1, \tau_2, \ldots, \tau_q\}$).

La preuve du théorème dans ce cas est toujours basée sur l'analyse de la charge processeur afin de montrer que le système est dans le même état aux instants t_c+1 et t_c+T+1. La contribution à la charge processeur de chaque tâche τ_i de PT entre les instants t_c+1 et t_c+T+1 est $(\frac{T}{T_i}-1)C_i+RCT_i(t_c+1)+\overline{RCT_i}(t_c+T+1)$. Cela correspond à la charge des instances complètes, plus la charge restante de l'instance courante à t_c+1, plus la charge exécutée de l'instance courante à t_c+T+1. L'équation de la charge processeur totale est donc :

$W(t_c+1,t_c+T+1)=\sum_{\tau_i\in RE}\frac{T}{T_i}C_i$
$+\sum_{\tau_i\in PT}(\frac{T}{T_i}C_i+\left[RCT_i(t_c+1)+\overline{RCT}_i(t_c+T+1)-C_i\right])$
$+\sum_{\tau_i\in LR}(\left\lfloor\frac{t_c+T+1-r_i}{T_i}\right\rfloor C_i+\overline{RCT_i}(t_c+T+1))$
$+\sum_{\tau_i\in AR}((\frac{T}{T_i}-1)C_i+\overline{RCT_i}(t_c+T+1))$
$=mT.$

La preuve du théorème est très proche de celle du cas du temps creux complet. Mais dans un premier temps nous montrons que le lemme 2 reste valide. La preuve est presque la même que précédemment : en effet, pour le calcul de la charge processeur, il suffit d'ajouter la contribution à la charge totale des tâches de PT. La contribution de chaque tâche τ_i de PT est $\frac{T}{T_i}C_i$ ($=T\frac{C_i}{T_i}$ qui rentre dans le calcul de UT), plus $RCT_i(t_c+1)+\overline{RCT_i}(t_c+T+1)-C_i$ qui est inférieur ou égal à 0 de par la propriété de monotonie. Le reste de la preuve se déduit du cas précédent.

Nous avons alors :

$mT=W(t_c+1,t_c+T+1)$
$=\sum_{\tau_i\in RE}\frac{T}{T_i}C_i$
$+\sum_{\tau_i\in PT}(\frac{T}{T_i}C_i+\left[RCT_i(t_c+1)+\overline{RCT}_i(t_c+T+1)-C_i\right])$
$+\sum_{\tau_i\in LR}(\left\lfloor\frac{t_c+T+1-r_i}{T_i}\right\rfloor C_i+\overline{RCT_i}(t_c+T+1))$
$+\sum_{\tau_i\in AR}((\frac{T}{T_i}-1)C_i+\overline{RCT_i}(t_c+T+1))$
$=mT+\sum_{\tau_i\in PT}(RCT_i(t_c+1)+\overline{RCT}_i(t_c+T+1)-C_i)$
$+\sum_{\tau_i\in LR}(\overline{RCT_i}(t_c+T+1)-C_i)$
$+\sum_{\tau_i\in AR}(\overline{RCT_i}(t_c+T+1)-C_i).$

Compte tenu de la monotonie de l'ordonnancement, $RCT_i(t_c+1)+\overline{RCT}_i(t_c+T+1)-C_i\leq 0,\forall\tau_i\in PT$. D'autre part, $\overline{RCT_i}(t_c+T+1)-C_i\leq 0,\forall\tau_i\in LR\cup AR$. Il s'en suit alors

que : $RCT_i(t_c+1)+\overline{RCT}_i(t_c+T+1)-C_i=0, \forall\tau_i \in PT$ et
$\overline{RCT_i}(t_c+T+1)-C_i=0, \forall\tau_i \in LR \cup AR$. On déduit alors la preuve du théorème comme dans le premier cas.□

3.4 La classe des ordonnancements monotones

Nous avons caractérisé l'entrée dans le cycle pour les ordonnancement monotones, déterministes et pseudo-conservatifs. Nous montrons maintenant que la classe des ordonnancements monotones est assez large, et en particulier nous montrons dans la suite du document qu'elle contient plusieurs stratégies d'ordonnancement classiques.

3.4.1 Ordonnancements à priorités fixes

Nous supposons que des priorités fixes sont assignées aux tâches et que deux tâches distinctes ont des priorités différentes.

Proposition 2. *Les ordonnancements à priorités fixes sont des ordonnancements monotones.*

Preuve :
Considérons un ordonnancement O à priorités fixes. Soit τ_i une tâche, et t un instant tel que $t \geq r_i$. Nous devons prouver que $RCT_i(t+T) \geq RCT_i(t)$ ou de manière équivalente établir que $\overline{RCT}_i(t+T) \leq \overline{RCT}_i(t)$. Nous considérons trois cas :

1- l'instant t est un instant d'activation d'une instance de τ_i. Alors, $RCT_i(t) = C_i$. $t+T$ est aussi un instant d'activation de τ_i, et donc $\overline{RCT_i}(t+T) = 0$, d'où la propriété de monotonie.

2- t n'est pas un instant d'activation de τ_i et $PI(t)$ s'est exécutée sans préemption depuis son activation. Soit $r_i + kT_i$ la date d'activation de l'instance. Alors, $\overline{RCT_i}(t) = t-(r_i+kT_i)$ et nécessairement, $\overline{RCT_i}(t+T) \leq (t+T)-(r_i+T+kT_i)$.

Donc, $\overline{RCT_i}(t+T) \leq \overline{RCT_i}(t)$, d'où la propriété de monotonie.

3- l'instant t ne correspond pas à une date d'activation d'une instance de la tâche τ_i et l'exécution de $PI(t)$ n'a pas été continue. Cela veut dire qu'il y a eu préemption par des tâches plus prioritaires. Pour traiter ce cas, nous allons introduire quelques notions supplémentaires.

Définition 17. 1. *Un instant t est appelé point de préemption s'il existe au moins une tâche active qui n'est pas ordonnancée à cet instant.*

2. *si t est un point de préemption, son contexte de préemption est un* $(m+1)-uplet$ $Ctx(t) = (\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}, list)$ *tel que :*
   – $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ *sont les tâches ordonnancées à l'instant t et*
   – $list$ *est l'ensemble des tâches en attente à cet instant.*

Nous pouvons tout d'abord noter que si t est un point de préemption et $Ctx(t)$ son contexte, $Ctx(t).list$ n'est pas vide. De plus, si $\tau_k \in Ctx(t).list$, alors la priorité de toute tâche $\tau_{i_j}, j = 1, \ldots, m$ est supérieure à la priorité de τ_k.
Nous notons $\left(tp_1, tp_2, \ldots, tp_f\right)$ la suite croissante des points de préemption avant t_c. Puisque t_c est un temps creux, alors il n'est pas un point de préemption. Nous prouvons maintenant qu'une tâche active qui n'a pas été ordonnancée à un instant t ne le sera pas à l'instant $t+T$.

Lemme 3. *Soit t_p un point de préemption et τ_k une tâche telle que $\tau_k \in Ctx(t_p).list$; alors τ_k n'est pas ordonnancée à l'instant $t_p + T$, c'est-à-dire $O_k(t_p+T) = 0$.*

Preuve du lemme :
Nous procédons par induction sur les points de préemption.
I - Considérons le premier point de préemption t_{p_1} et son contexte

$$Ctx(t_{p_1}) = \{\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}, list_1\}.$$

Soit τ_k une tâche quelconque dans $list_1$. Nous considérons les trois cas suivant :

1. $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ ont été activées avant t_{p_1}. Elles se sont donc exécutées continuellement sans préemption et ont une priorité supérieure à τ_k. Puisque t_{p_1} est le premier point de préemption, l'instance $PI_k(t_{p_1})$ de la tâche τ_k a donc été activée exactement à t_{p_1}. D'autre part, nous avons $\overline{RCT}_{i_j}(t_{p_1}) \geq \overline{RCT}_{i_j}(t_{p_1}+T)$ (parce que l'instance courante de τ_{i_j} a été continue sans préemption). De plus, puisqu'elle a été ordonnancée à l'instant t_{p_1}, alors τ_{i_j} n'a pas fini son instance courante à t_{p_1} et donc elle n'aura pas non plus fini son l'instance à $t_{p_1}+T$. Par suite, les tâches $\tau_{i_j}, j=1,\ldots,m$ sont toujours en cours d'exécution à $t_{p_1}+T$, plus prioritaires que τ_k, donc τ_k ne pourra pas être ordonnancée à $t_{p_1}+T$.
2. Supposons maintenant que certaines tâches τ_{i_j} se sont activées avant t_{p_1} et d'autres à t_{p_1}. Les tâches qui sont activées à t_{p_1} le sont aussi à $t_{p_1}+T$. Soit τ_{i_u} une tâche qui s'est activée avant t_{p_1}. Son instance a été exécutée continuellement sans préemption de son activation à t_{p_1}, donc
   $\overline{RCT_{i_u}}(t_{p_1})$ (qui est maximal)$\geq \overline{RCT_{i_u}}(t_{p_1}+T)$.
   Ici encore toutes les tâches τ_{i_j} $(j=1,\ldots,m)$ sont toujours en cours d'exécution à $t_{p_1}+T$ et nous concluons comme précédemment.

□

II - Nous proposons d'abord le corollaire suivant.

Corollaire 1. *Supposons le lemme vérifié pour* $\{t_{p_1}, t_{p_2}, \ldots, t_{p_s}\}$ *c'est à dire que*

$$\forall t_p \in \{t_{p_1}, t_{p_2}, \ldots, t_{p_s}\}, O_k(t_p+T) = 0.$$

Alors, $\forall t \leq t_{p_{s+1}}, \forall \tau_k$ *telle que* $r_k \leq t, \overline{RCT}_k(t) \geq \overline{RCT}_k(t+T)$.

En effet, à chaque fois que τ_k est mise en attente à un instant t_{p_u}, elle l'est aussi à l'instant $t_{p_u}+T$. La tâche τ_k est donc inactive au moins aussi souvent entre le début de l'instance $PI_k(t+T)$ et $t+T$ qu'entre le début de l'instance $PI_k(t)$ et t. Il s'en suit alors que $\overline{RCT}_k(t) \geq \overline{RCT}_k(t+T)$.

III - Nous nous plaçons à l'instant $t_{p_{s+1}}$ La fin de la preuve du lemme correspond corolaire avec $t = t_{p_{s+1}}$ et en considérant une tâche τ_{j_k} du contexte de $t_{p_{s+1}}$. Nous avons donc $\overline{RCT}_{j_k}(t_{p_{s+1}}) \geq \overline{RCT}_{j_k}(t_{p_{s+1}} + T)$. Et comme précédemment, nous concluons qu'aucune tâche du contexte de $t_{p_{s+1}}$ ne peut avoir terminé son exécution à $t_{p_{s+1}} + T$, et donc τ_k, qui a une priorité moins élevée, ne pourra s'exécuter à l'instant $t_{p_{s+1}} + T$.□

Les ordonnancements à priorités fixes sont donc des ordonnancements monotones d'après le corollaire appliqué avec $k = i$.□

3.4.2 Cas des ordonnancements EDF

Nous considérons maintenant les ordonnancement *EDF* : les tâches ordonnancées sont celles dont les échéances sont les plus proches [81]. Nous supposons de plus qu'il existe un mécanisme déterministe de gestion des conflits. Par exemple en cas d'échéances identiques, la priorité est donnée à la tâche de plus petit numéro.

Proposition 3. *Les ordonnancements EDF avec gestion déterministe des conflits sont monotones.*

Preuve :
Ici aussi, nous devons prouver que $\forall \tau_i$ et $\forall t$ tels que $t \geq r_i$, $RCT_i(t+T) \geq RCT_i(t)$ ou de manière équivalente $\overline{RCT}_i(t+T) \leq \overline{RCT}_i(t)$.
Supposons que ce résultat ne soit pas valide. Nous définissons alors t_0 comme le premier instant t où une tâche τ_i vérifie $\overline{RCT}_i(t) < \overline{RCT}_i(t+T)$ (avec $t \geq r_i$). Soit τ_{i_0} une telle tâche. Nous avons :

$$\begin{cases} \overline{RCT}_{i_0}(t_0) < \overline{RCT}_{i_0}(t_0+T) & (1) \\ \overline{RCT}_{i_0}(t_0-1) = \overline{RCT}_{i_0}(t_0+T-1) & (2) \end{cases}$$

Nous déduisons de (1) que t_0 n'est pas un instant d'activation de la tâche τ_{i_0} (sinon nous aurions $\overline{RCT}_{i_0}(t_0) = 0 = \overline{RCT}_{i_0}(t_0+T)$) et τ_{i_0} est ordonnancée à l'instant $t_0 + T - 1$ mais pas à l'instant $t_0 - 1$. Nous déduisons aussi que la tâche τ_{i_0} n'a pas fini son instance à $t_0 - 1$. Par suite, du fait que l'ordonnancement est conservatif, il y a m tâches $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ qui sont ordonnancées à l'instant $t_0 - 1$. Leur exécution n'est

donc pas finie à cet instant et du fait qu'elles sont ordonnancées au détriment de τ_{i_0}, leurs échéances sont plus proches que celle de τ_{i_0} ou, en cas d'égalité, le mécanisme de gestion des conflits leur donne la priorité. D'après la définition de l'instant t_0, les tâches $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ vérifient $\overline{RCT}_{i_j}(t_0+T-1) \leq \overline{RCT}_{i_j}(t_0-1)$. Donc à t_0+T-1 leurs instances courantes n'auront pas encore terminé leur exécution et leurs échéances seront plus proches que ou égales à celle de τ_{i_0}. En cas d'égalité des échéances de τ_{i_j} et τ_{i_0}, le déterminisme de la règle de gestion des conflits donne à nouveau la priorité à τ_{i_j}. Donc τ_{i_0} ne sera pas ordonnancée à t_0+T-1. Il s'en suit alors une contradition, d'où le résulat.□

3.4.3 Cas des ordonnancements LLF

Ici les tâches ordonnancées sont celles ayant la plus petite laxité. La laxité d'une tâche τ_i à un instant t que nous notons $Lax(\tau_i,t)$ est égale à la différence entre l'échéance de $PI_i(t)$ et $RCT_i(t)$: $Lax(\tau_i,t) = \textit{échéance de } PI_i(t) - RCT_i(t)$ [90]. Là encore, nous supposons qu'il existe un mécanisme déterministe de gestion des conflits.

Proposition 4. *LLF avec gestion déterministe des conflits est un ordonnancement monotone.*

Preuve :

Ici aussi prouvons qu'une tâche τ_i ne peut pas recevoir plus de temps processeur à l'instant $t+T$ qu'à l'instant t ($t \geq r_i$). Formellement, nous prouvons que :
$\forall t, \forall \tau_i, t \geq r_i \Rightarrow \overline{RCT_i}(t) \geq \overline{RCT_i}(t+T)$.
La démarche adoptée pour la preuve est la même que dans le cas des ordonnancements EDF. Supposons que le résultat ne soit pas valide. Alors notons, t_0 le premier instant pour lequel il existe une tâche τ_i vérifiant $\overline{RCT_i}(t_0) < \overline{RCT_i}(t_0+T)$. Soit τ_{i_0} une telle tâche. Alors nous avons $\begin{cases} \overline{RCT_{i_0}}(t_0) < \overline{RCT_{i_0}}(t_0+T) \ \ (1) \\ \overline{RCT_{i_0}}(t_0-1) = \overline{RCT_{i_0}}(t_0+T-1) \ \ (2) \end{cases}$ Nous en déduisons que t_0 n'est pas une date d'activation pour τ_{i_0} (sinon on aurait $\overline{RCT}_{i_0}(t_0) = \overline{RCT}_{i_0}(t_0+T) = 0$) et que τ_{i_0} a été ordonnancée à t_0+T-1 mais pas à t_0-1. Nous en déduisons aussi que τ_{i_0} n'a pas terminé son exécution à t_0-1. Alors, l'ordonnancement LLF étant

conservatif, il y a m tâches notées $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ qui sont ordonnancées à t_0. D'après la définition de l'instant t_0, nous avons (3) $\overline{RCT_{i_j}}(t_0-1) \geq \overline{RCT_{i_j}}(t_0+T-1)$ et d'après les propriétés de l'ordonnancement LLF, nous avons (4) $Lax(\tau_{i_j}, t_0-1) \leq Lax(\tau_{i_0}, t_0-1)$.
De plus, de (2), nous tirons $Lax(\tau_{i_0}, t_0+T-1) = Lax(\tau_{i_0}, t_0-1)+T$. De (3) nous avons $Lax(\tau_{i_j}, t_0+T-1) \leq Lax(\tau_{i_j}, t_0-1)+T$.
Combiné avec (4) cela donne $Lax(\tau_{i_j}, t_0+T-1) \leq Lax(\tau_{i_0}, t_0+T-1)$.
Compte tenu du déterminisme de gestion des conflits, les tâches $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ sont alors à nouveau plus prioritaires que τ_0 à t_0+T-1. Donc τ_0 ne peut pas avoir été ordonnancée à t_0+T-1, d'où la contradiction. Par suite la proposition est vérifiée pour l'ordonnancement LLF. $\square$

3.4.4 Cas des ordonnancements P-équitables

Nous considérons dans cette partie les ordonnancements PF et PD^2. Nous supposons que les ex-æquo sont départagés de façon déterministe. Par exemple, la priorité sera donnée à la tâche de plus petit numéro.

Proposition 5. *Les algorithmes PF et PD^2 avec gestion déterministe des ex-æquo sont des algorithmes monotones.*

Preuve
Dans les deux preuves, nous allons raisonner par l'absurde. Nous voulons montrer que $\forall \tau_i$, $\forall t \geq r_i$, $\overline{RCT}_i(t) \geq \overline{RCT}_i(t+T)$.
Nous supposons le contraire. Il existe donc un instant t_0 qui est le premier instant où il existe au moins une tâche τ_{i_0} telle que $\overline{RCT}_{i_0}(t_0) < \overline{RCT}_{i_0}(t_0+T)$. Nous avons donc

$$\begin{cases} \overline{RCT}_{i_0}(t_0) < \overline{RCT}_{i_0}(t_0+T) & (1) \\ \overline{RCT}_{i_0}(t_0-1) = \overline{RCT}_{i_0}(t_0-1+T) & (2) \end{cases}$$

La tâche τ_{i_0} est donc exéctuée à l'instant t_0+T-1 mais pas à l'instant t_0-1.

1 - Nous considérons tout d'abord l'algorithme PF. Nous revenons aux définitons initiales issues de [18]. Considérons les sous-chaîne caractéristiques. L'ensemble des sous-

chaînes caractéristiques est muni de la relation d'ordre lexicographique (avec $- \prec 0 \prec +$). Dans ce cas, l'algorithme PF peut être décrit de la façon suivante [18] :

1. Ordonnancer les tâches urgentes (soit k leur nombre).
2. Trier les tâches possibles par ordre lexicographique des sous-chaînes caractéristiques.
3. Ordonnancer les k premières d'entre elles. En cas d'ex-aequo, on utilise la règle déterministe de départage.

Nous utilisons cette description pour montrer que PF est monotone.

Lemme 4. *Les sous chaînes caractéristiques de τ_i sont périodiques de période T_i.*

Ceci découle directement des définitions.

Nous supposons t_0 et τ_{i_0} définis. Nous discutons selon le statut de la tâche τ_{i_0} à l'instant $t_0 - 1$.

• La tâche n'est pas urgente à l'instant $t_0 - 1$, sinon elle aurait été exécutée.

• Si elle est interdite, cela signifie que $W_{i_0}(0, t_0 - 1) + 1 \geq u_{i_0} \times t_0 + 1$ (si la tâche s'exécutait, sa courbe de charge se retrouverait au dessus de la droite limite supérieure). Mais dans ce cas, d'après l'égalité (2), nous avons :
$W_{i_0}(0, t_0 + T - 1) + 1 = W_{i_0}(0, t_0 - 1) + 1 + \frac{T}{T_{i_0}} \times C_{i_0}$ et donc nous avons
$W_{i_0}(0, t_0 + T - 1) + 1 \geq u_{i_0} \times t_0 + 1 + \frac{T}{T_{i_0}} \times C_{i_0} = u_{i_0} \times (t_0 + T) + 1$. Par suite, τ_{i_0} est aussi interdite à l'instant $t_0 + T - 1$. Ce qui contredit notre hypothèse.

• La tâche τ_{i_0} est donc possible à l'instant $t_0 - 1$. Elle ne s'est pas exécutée, donc il existe m tâches $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ qui sont exécutées. Elles sont donc, soit urgentes, soit possibles.

– Si une tâche τ_{i_j} est urgente à $t_0 - 1$, nous avons $W_{i_j}(0, t_0 - 1) \leq u_{i_j} \times t_0 - 1$ (si la tâche ne s'exécute pas, sa courbe d'exécution passe en dessous de la droite limite inférieure). Par ailleurs, d'après la définition de t_0, nous avons
$\overline{RCT}_{i_j}(t_0 - 1) \geq \overline{RCT}_{i_j}(t_0 + T - 1)$ donc nous avons
$W_{i_j}(0, t_0 - 1) + \frac{T}{T_{i_j}} \times C_{i_j} \geq W_{i_j}(0, t_0 + T - 1)$. Nous en déduisons donc que
$W_{i_j}(0, t_0 + T - 1) \leq u_{i_j} \times t_0 - 1 + \frac{T}{T_{i_j}} \times C_{i_j} = u_{i_j} \times (t_0 + T) - 1$. La tâche τ_{i_j} est donc également urgente à l'instant $t_0 + T - 1$.

– Si une tâche τ_{i_j} est possible, elle est plus prioritaire que τ_{i_0}. Cela signifie :

◇ soit que sa sous-chaîne caractéristique est inférieure à celle de τ_{i_0} : $\alpha(i_j, t_0 - 1) \prec \alpha(i_0, t_0 - 1)$;

◇ soit que les sous-chaînes sont égales, et τ_{i_j} est plus prioritaire d'après la règle de départage des ex-aequo.

A l'instant $t_0 + T - 1$, on a $\overline{RCT}_{i_j}(t_0 - 1) \geq \overline{RCT}_{i_j}(t_0 + T - 1)$, donc soit τ_{i_j} est urgente, et elle est donc exécutée, soit elle est possible, et comme d'après le lemme 4, les sous-chaînes caractéristiques sont périodiques, nous avons donc $\alpha(i_j, t_0 + T - 1) \prec \alpha(i_0, t_0 + T - 1)$ ou bien $\alpha(i_j, t_0 + T - 1) = \alpha(i_0, t_0 + T - 1)$ et τ_{i_j} plus prioritaire que τ_{i_0} (à cause du déterminisme de la règle de départage des ex-aequo). Donc τ_{i_j} est à nouveau plus prioritaire que τ_{i_0}.

Par suite, τ_{i_0} ne peut pas s'exécuter à l'instant $t_0 + T - 1$ ce qui contredit notre hypothèse. Il s'ensuit que l'on a $\forall \tau_i$, $\forall t \geq r_i$, $\overline{RCT}_i(t) \geq \overline{RCT}_i(t + T)$. L'algorithme PF est donc monotone.

2 - Considérons maintenant l'algorithme PD^2. Nous considérons l'algorithme tel qu'il a été présenté à la page 42. Nous rappelons que τ_i^j désigne la $i^{ème}$ sous-tâche unitaire de τ_i, r_i^j désigne sa pseudo date d'activation et d_i^j sa pseudo-échéance.

Considérons un instant t et une tâche τ_i. Nous notons $j_i(t)$ le numéro de la prochaine sous-tâche de τ_i à exécuter. La sous-tâche de numéro $j_i(t) - 1$ a déjà été exécutée, mais pas la sous-tâche numéro $j_i(t)$. Nous avons donc $j_i(t) = W_i(0, t) + 1$.

Nous supposons t_0 et τ_{i_0} définis.

Nous avons $\overline{RCT}_{i_0}(t_0 - 1) = \overline{RCT}_{i_0}(t_0 + T - 1)$.

Nous en déduisons que $W_{i_0}(0, t_0 - 1) + \frac{T}{T_{i_0}} \times C_{i_0} = W_{i_0}(0, t_0 + T - 1)$ et par suite :

(i) $j_{i_0}(t_0 + T - 1) = j_{i_0} + \frac{T}{T_{i_0}} \times C_{i_0}$

(ii) $d_{i_0}^{j_0(t_0+T-1)} = d_{i_0}^{j_0(t_0-1)} + T$

(iii) $r_{i_0}^{j_0(t_0+T-1)} = r_{i_0}^{j_0(t_0-1)} + T$

(iv) $b_{i_0}^{j_0(t_0+T-1)} = b_{i_0}^{j_0(t_0-1)}$

(v) L'échéance de groupe de $\tau_{i_0}^{j_0(t_0+T-1)}$ est égale à l'échéance de $\tau_{i_0}^{j_0(t_0-1)} + T$: $D_{i_0}^{j_0(t_0+T-1)} = D_{i_0}^{j_0(t_0-1)} + T$

Nous avons de plus $\overline{RCT}_{i_0}(t_0) < \overline{RCT}_{i_0}(t_0+T)$ donc

$$\begin{cases} \tau_{i_0}^{j_{i_0}(t_0-1)} \; n'est\ pas\ exécutée\ à\ l'instant\ t_0-1. \\ \tau_{i_0}^{j_{i_0}(t_0+T-1)} \; est\ exécutée\ à\ l'instant\ t_0+T-1 \end{cases}$$

$\tau_{i_0}^{j_{i_0}(t_0-1)}$ ne s'exécute à l'instant t_0-1 pas si

- $t_0-1 < r_{i_0}^{j_{i_0}(t_0-1)}$.
  Mais dans ce cas, on aurait $t_0+T-1 < r_{i_0}^{j_{i_0}(t_0-1)} + T = r_{i_0}^{j_{i_0}(t_0+T-1)}$ et donc $\tau_{i_0}^{j_{i_0}}(t_0+T-1)$ n'aurait pas pu s'exécuter à l'instant t_0+T-1.

- Il existe m sous tâches plus prioritaires. Soit $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ leurs tâches mères (τ_i est la tâche mère des sous-tâches τ_i^j).
  Par définition de t_0, nous avons $\overline{RCT}_{i_j}(t_0-1) \geq \overline{RCT}_{i_j}(t_0+T-1)$.
  Par suite, $j_{i_k}(t_0+T-1) \leq j_{i_k}(t_0-1) + \frac{T}{T_{i_k}} \times C_{i_k}$, et donc
  (A) $d_{i_k}^{j_{i_k}(t_0+T-1)} \leq d_{i_k}^{j_{i_k}(t_0-1)} + T$.
  Une sous-tâche $\tau_{i_k}^{j_k(t_0-1)}$ est plus prioritaire que $\tau_{i_0}^{j_{i_0}(t_0-1)}$ si :
  – $d_{i_k}^{j_{i_k}(t_0-1)} < d_{i_0}^{j_0(t_0-1)}$.
    Dans ce cas, d'après (A) et (ii),
    $d_{i_k}^{j_{i_k}(t_0+T-1)} \leq d_{i_k}^{j_{i_k}(t_0-1)} + T < d_{i_0}^{j_{i_0}(t_0-1)} + T = d_{i_0}^{j_{i_0}(t_0+T-1)}$. Donc $\tau_{i_k}^{j_{i_k}(t_0+T-1)}$ est également plus prioritaire que $\tau_{i_0}^{j_{i_0}(t_0+T-1)}$.
  – $d_{i_k}^{j_{i_k}(t_0-1)} = d_{i_0}^{j_0(t_0-1)}$ et $b_{i_k}^{j_{i_k}(t_0-1)} > b_{i_0}^{j_0(t_0-1)}$.
    Dans ce cas,
    • soit $j_{i_k}(t_0+T-1) = j_{i_k}(t_0-1) + \frac{T}{T_{i_k}} \times C_{i_k}$.
    Nous avons alors $d_{i_j}^{j_j(t_0+T-1)} = d_{i_0}^{j_0(t_0+T-1)}$. Nous avons également $b_{i_k}^{j_{i_k}(t_0+T-1)} = b_{i_k}^{j_{i_k}(t_0-1)} > b_{i_0}^{j_0(t_0-1)} = b_{i_0}^{j_0(t_0+T-1)}$.
    Donc $\tau_{i_k}^{j_{i_k}(t_0+T-1)}$ est plus prioritaire que $\tau_{i_0}^{j_{i_0}(t_0+T-1)}$.
    • Soit $j_{i_k}(t_0+T-1) < j_{i_k}(t_0-1) + \frac{T}{T_{i_k}} \times C_{i_k}$.
    Dans ce cas, $d_{i_k}^{j_{i_k}(t_0+T-1)} < d_{i_0}^{j_{i_0}(t_0+T-1)}$. Donc $\tau_{i_k}^{j_{i_k}(t_0+T-1)}$ est plus prioritaire que $\tau_{i_0}^{j_{i_k}(t_0+T-1)}$
  – $d_{i_k}^{j_{i_k}(t_0-1)} = d_{i_0}^{j_0(t_0-1)}$, $b_{i_k}^{j_{i_k}(t_0-1)} = b_{i_0}^{j_0(t_0-1)} = 1$ et $D_{i_k}^{j_{i_k}(t_0-1)} > D_{i_0}^{j_0(t_0-1)}$. Alors,

• soit $j_{i_k}(t_0+T-1) = j_{i_k}(t_0-1) + \frac{T}{T_{i_k}} \times C_{i_k}$, dans ce cas, $d_{i_k}^{j_{i_k}(t_0+T-1)} = d_{i_0}^{j_{i_0}(t_0+T-1)}$ et $b_{i_k}^{j_{i_k}(t_0+T-1)} = b_{i_k}^{j_{i_k}(t_0-1)} = b_{i_0}^{j_{i_0}(t_0-1)} = b_{i_0}^{j_{i_0}(t_0+T-1)} = 1$. Par ailleurs, on a $D_{i_k}^{j_{i_k}(t_0+T-1)} = D_{i_k}^{j_{i_k}(t_0-1)} + T > D_{i_0}^{j_{i_0}(t_0-1)} + T = D_{i_0}^{j_{i_0}(t_0+T-1)}$. Il s'ensuit que $\tau_{i_k}^{j_{i_k}(t_0+T-1)}$ est également plus prioritaire que $\tau_{i_0}^{j_{i_0}(t_0+T-1)}$.

• Soit $j_{i_k}(t_0+T-1) < j_{i_k}(t_0-1) + \frac{T}{T_{i_k}} \times C_{i_k}$. Dans ce cas, $d_{i_k}^{j_{i_k}(t_0+T-1)} < d_{i_0}^{j_{i_0}(t_0+T-1)}$. Donc $\tau_{i_k}^{j_{i_k}(t_0+T-1)}$ est plus prioritaire que $\tau_{i_0}^{j_{i_0}(t_0+T-1)}$

– $d_{i_k}^{j_{i_k}(t_0-1)} = d_{i_0}^{j_0(t_0-1)}$, $b_{i_k}^{j_{i_k}(t_0-1)} = b_{i_0}^{j_0(t_0-1)} = 1$ et $D_{i_k}^{j_{i_k}(t_0-1)} = D_{i_0}^{j_0(t_0-1)}$.
La règle de départage des ex-aequo est donc utilisée. Alors
• soit $j_{i_k}(t_0+T-1) = j_{i_k}(t_0-1) + \frac{T}{T_{i_k}} \times C_{i_k}$, dans ce cas, $d_{i_k}^{j_{i_k}(t_0+T-1)} = d_{i_0}^{j_{i_0}(t_0+T-1)}$ et $b_{i_k}^{j_{i_k}(t_0+T-1)} = b_{i_k}^{j_{i_k}(t_0-1)} = b_{i_0}^{j_{i_0}(t_0-1)} = b_{i_0}^{j_{i_0}(t_0+T-1)} = 1$ et $D_{i_k}^{j_{i_k}(t_0+T-1)} = D_{i_0}^{j_0(t_0+T-1)}$.
Les tâches sont à nouveau ex-aequo, et le départage se fait à nouveau en faveur de τ_{i_k} à cause du déterminisme.
• Soit $j_{i_k}(t_0+T-1) < j_{i_k}(t_0-1) + \frac{T}{T_{i_k}} \times C_{i_k}$.
Dans ce cas, $d_{i_k}^{j_{i_k}(t_0+T-1)} < d_{i_0}^{j_{i_0}(t_0+T-1)}$. Donc $\tau_{i_k}^{j_{i_k}(t_0+T-1)}$ est plus prioritaire que $\tau_{i_0}^{j_{i_0}(t_0+T-1)}$

Par conséquent, les m tâches $\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}$ seront prioritaires sur τ_{i_0} à l'instant t_0+T-1.

Donc dans tous les cas de figure, τ_{i_0} ne pourra pas s'exécuter à cet instant, ce qui contredit notre hypothèse. Par conséquent, il n'existe pas de couples (t_0, τ_{i_0}) vérifiant les propriétés (1) et (2). La propriété de monotonie est donc vérifiée.□

3.5 Date du dernier temps creux acycliques

Les résultats sur la monotonie nous permettent de conclure que notre résultat sur la cyclicité est valide pour les ordonnancements à priorités fixes, pour les ordonnancements selon une politique *EDF* et *LLF* et enfin pour les ordonnancements P-équitables *PF* et

PD^2.

Cependant la recherche d'une borne supérieure pour t_c reste une question ouverte. Dans le cas monoprocesseur, les travaux de [37] ont montré que cette borne est $r+T$ quel que soit l'algorithme d'ordonnancement déterministe.

Dans le cas multiprocesseur, les résultats de nos simulations ont montré que cette borne peut être très grande comme le montrent les figures 3.5 et 3.6. La figure 3.5 donne la séquence d'ordonnancement $O1$ du système de tâches

$$\tau = \{\tau_1(5,6,11,11), \tau_2(0,6,11,11), \tau_3(0,6,11,11), \tau_4(3,4,11,11)\}$$

sur deux processeurs. Sur la séquence, le 0 représente un temps creux. $O1$ est la séquence EDF. Sur cette séquence, le cycle commence à l'instant 55 après un temps creux à l'instant 54. Nous avons $max\{r_i\}_{i=1\ldots,4} = r = 5$. Nous pouvons donc constater que $54 = 5+4\times 11+5 = r+4T+5 > r+T$.

Avec le système de tâches

$$\tau = \{\tau_1(225,90,161,161), \tau_2(115,40,161,161),$$

$$\tau_3(0,72,161,161), \tau_4(129,120,161,161),$$

le dernier temps creux acyclique apparaît à $7037 = 2+42T+50$. Donc, dans un contexte d'ordonnancement EDF, la date d'entrée dans le cycle peut être très éloignée.

Il peut en être de même avec la politique d'ordonnancement LLF. Sur la figure 3.6, $O2$ est la séquence d'ordonnancement LLF sur deux processeurs du système de tâches

$$\tau = \{\tau_1(5,4,11,11), \tau_2(0,6,11,11), \tau_3(4,6,11,11), \tau_4(3,6,11,11)\}.$$

Sur cette séquence le cycle commence à 26. Le dernier temps creux acyclique apparaît ici à $25 = r+T+9$.

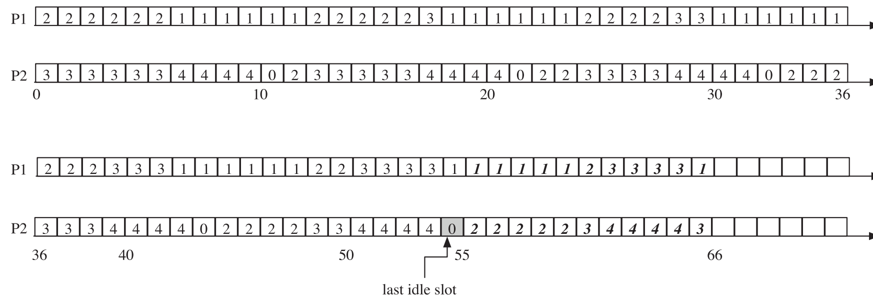


Figure 3.5 – La cyclicité sous *EDF* :
la figure montre la séquence d'ordonnancement suivant l'algorithme EDF sur 2 processeurs du système de tâches $\tau = \{\tau_1(5,6,11,11), \tau_2(0,6,11,11), \tau_3(0,6,11,11), \tau_4(3,4,11,11)\}$. *Le cycle commence à l'instant* $t = 55$.

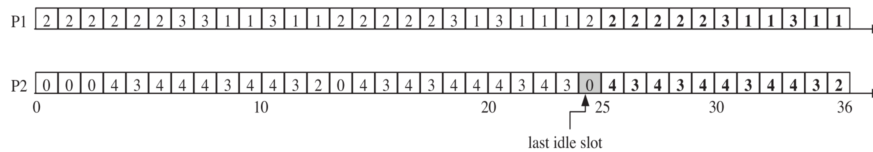


Figure 3.6 – La cyclicité sous *LLF* :
la figure montre la séquence d'ordonnancement suivant un algorithme LLF sur 2 processeurs du système de tâches
$\tau = \{\tau_1(5,4,11,11), \tau_2(0,6,11,11), \tau_3(4,6,11,11), \tau_4(3,6,11,11)\}$. *Le cycle commence à l'instant t=26.*

Pour les ordonnancement à priorités fixes ainsi que les ordonnancements équitables selon une politique *PF*, les choses semblent cependant différentes. Nous avons constaté que pour les ordonnancements à priorités fixes, la borne $r + T$ est restée valide en multiprocesseur comme l'illustre le tableau 3.I. Nous avons généré des échantillons de 1000 tâches. Ces échantillons sont générés de sorte que pour chaque système de tâches nous ayons $U \leq m$. Pour chaque architecture nous génèrons donc un échantillon de 1000 tâches périodiques selon l'approche décrite à la page 63.

RM		EDF		LLF	
$t_c \leq r+T$	$t_c > r+T$	$t_c \leq r+T$	$t_c > r+T$	$t_c \leq r+T$	$t_c > r+T$
100 %	0 %	86,21 %	13,78 %	86,67 %	13,33 %

Tableau 3.I – Date d'entrée dans le cycle sur deux processeurs et selon les algorithmes RM, EDF et LLF

Il apparait également, d'après les simulations résumés dans le tableau 3.II que cette borne reste valable quand on considère les ordonnancements équitables selon une politique PF.

	Nombre de processeurs					
Systèmes de tâches	2	3	4	5	6	%
$0 \leq t_0 \leq max(r_i)$	0	0	0	0	0	0
$t_0 = max(r_i)$	852	741	721	689	705	74.6
$max(r_i) < t_0 \leq max(r_i)+T$	148	259	279	311	295	25.84
$t_0 > max(r_i)+T$	0	0	0	0	0	0

Tableau 3.II – Résultats de la simulation sur la cyclicité. L'algorithme d'ordonnancement utilisé est PF. t_0 désigne le temps d'entrée dans le cycle.

Conjecture 1. *Nous conjecturons donc que sur une plateforme multiprocesseur et pour des ordonnancements à priorités fixes et des ordonnancements P-équitables,* $t_c \leq r+T$.

3.6 Conclusion

Dans ce chapitre nous avons considéré l'ordonnancement des applications temps-réel sur des plateformes multiprocesseur. Nous avons présenté plus précisément le problème de la cyclicité dans le contexte des ordonnancement multiprocesseurs et nous avons caractérisé l'entrée dans le régime permanent pour les ordonnancements monotones déterministes. Nous avons montré que les ordonnancements à priorités fixes, les

ordonnancements à priorités dynamiques tels que *EDF* et *LLF* et enfin les ordonnancements P-équitables tels que *PF* et PD^2 sont des ordonnancements monotones.
Les simulations que nous avons effectuées nous ont permis de conjecturer que la date d'entrée dans le cycle est majorée par $max\{r_i\}_{i=1...n} + T$ dans le cadre des ordonnancement à priorités fixes et des ordonnancements P-équitables. Cependant cette même date d'entrée dans le cycle peut être très éloignée quand on considère les ordonnancements à priorités dynamiques. Il en resulte que toute borne générale ne pourra qu'être péssimiste. Ceci renforce donc l'intérêt de la détermination algorithmique de la date d'entrée dans le cycle sur la base de notre critère. Lorsqu'on manipule des ressources, celles-ci peuvent induire la création de temps creux complémentaires (ce qui n'était pas le cas pour les systèmes monoprocesseur). Le problème de la cyclicité reste donc toujours un problème ouvert dans ce contexte car jusque là aucun résultat n'a donné une borne à la date d'entrée dans la phase cyclique.

CHAPITRE 4

RECONFIGURATION EN CAS DE PANNE PROCESSEUR

Dans ce chapitre, nous nous intéressons au problème de la panne d'un processeur. Si les architectures multiprocesseur deviennent incontournables en réponse aux besoins toujours croissants en terme de puissance de calcul, elles présentent cependant un risque important en terme de robustesse, la probabilité qu'un processeur tombe en panne augmentant avec le nombre de processeurs. Il s'ensuit qu'il est nécessaire, lors de la conception de tels systèmes, de prévoir des mécanismes de détection de pannes, et de reconfiguration efficace : il s'agit de passer en-ligne d'un mode à m processeurs à un mode à $m-1$ processeurs, et ceci tout en minimisant les redondances afin de limiter les surcoûts engendrés.

4.1 Périmètre de l'étude

4.1.1 Nos hypothèses

Nous considérons des systèmes à m processeurs identiques, disposant d'une mémoire commune. Nous supposons qu'une panne peut affecter un processeur quelconque à tout moment. Nous supposons de plus que la panne n'affecte pas la mémoire globale. Par ailleurs, nous supposons qu'il existe un mécanisme de détection active de panne [29, 101]. Une fois la panne détectée, l'ordonnanceur ne placera plus aucune tâche sur le processeur en panne. Nous considérons un mécanisme de gestion des pannes par redondance : nous supposons qu'il y a un processeur supplémentaire, qui est utilisé au départ pour l'exécution de l'application. Pour cela, nous considérons donc des applications qui sont ordonnançable aussi bien sur $m-1$ processeurs que sur m. Elles s'exécutent au départ sur la totalité des m processeurs, puis, si une panne est détectée, elles se reconfigurent pour poursuivre leur exécution sur $m-1$ processeurs. Notre objectif est donc d'étudier le mécanisme de reconfiguration et de déterminer les cas où il est effectivement envisageable, c'est à dire où il ne produit aucune faute temporelle.

Dans ce qui suit, nous considérons des applications constituées de tâches périodiques indépendantes à départs simultanés, et nous envisageons les ordonnancements EDF et P-équitables (PF ou PD^2).

4.1.2 Les différents cas de figures

Nous avons supposé qu'il existait un mécanisme de détection active de panne. Nous adoptons l'hypothèse simplificatrice suivante : si une panne survient à l'instant t_p, elle est détectée à l'instant t_p. Dans ce cas, lorsqu'une panne survient, une tâche au plus va se trouver affectée : celle qui est en cours d'exécution sur le processeur qui subit la panne. Nous notons cette tâche τ_{i_0} dans la suite. Tout ou partie de cette tâche devra donc être ré-excécutée. Dans le cas où la panne n'est pas détectée instantanément (elle est détecté à un instant t' tel que $t' > t_p$), plusieurs tâches seront concernées : toutes celles qui ont été allouées au processeur en panne entre les instants t_p et t'. Nous ne traitons pas ce cas ici. Dans notre cas, de façon plus précise, au moment où survient la panne, divers scénarii sont envisageables :

S1 *La panne survient juste après une commutation de contexte*. Dans ce cas, l'application doit simplement basculer du mode m processeurs, au mode $m-1$ processeurs, mais aucune exécution n'est perdue. Il faudra donc juste s'assurer que la réorganisation est possible, et n'induit pas de dépassements d'échéances.

S2 *La panne survient pendant l'exécution de la tâche τ_{i_0}*. Dans ce cas, nous pouvons envisager 3 politiques :

P1 On ré-exécute uniquement ce qui a été exécuté depuis la dernière sauvegarde du contexte. Notons que pour permettre une telle reprise, il faut non seulement mémoriser le contexte processeur, mais aussi le contexte mémoire. Une telle sauvegarde peut être par exemple réalisée lors des changements de contexte. Cette politique permet de limiter le temps processeur requis pour la ré-exécution, mais en contrepartie, il est nécessaire de prévoir les durées requises par les sauvegarde. De plus, les changements de contexte ne sont pas répartis de manière régulière ni prévisible. La charge à exécuter est donc d'une taille non prévisible, et il est difficile de quantifier les surcoûts.

L'approche est encore appelée *approche par sauvegarde/restauration* et a été étudiée dans la littérature dans les travaux de [25, 58, 73, 89, 96, 102]. Le surcoût engendré par l'approche a été évalué et dépend du nombre de points de reprises de chaque tâche. Les travaux de Shin et al. [102] ont proposé une technique de placement optimal des points de reprise pour les systèmes temps-réel. Dans leur approche, les points de reprise sont régulièrement répartis sur chaque instance d'une tâche. Les travaux de [114] proposent que ces points de reprises soient équidistants sans toutefois donner le nombre optimal de points de reprises. Dans les travaux de [102], le nombre de points de reprises n_i à répartir sur une instance d'une tâche τ_i est :

$$n_i = \begin{cases} \left\lfloor \sqrt{\frac{C_i}{O}} \right\rfloor & si\ C_i \leq n_i(n_i+1)O \\ \\ \left\lceil \sqrt{\frac{C_i}{O}} \right\rceil & si\ C_i > n_i(n_i+1)O \end{cases}$$

O désigne le coût de sauvegarde du contexte de la tâche (ce coût est supposé constant pour toutes les tâches). Si n_i est calculé pour la tâche τ_i, alors l'état de la tâche est sauvé toutes les $\left\lceil \frac{C_i}{n_i} \right\rceil$ unités de temps.

En posant $R_i = \left\lceil \frac{C_i}{n_i} \right\rceil$, alors en cas de panne, le temps de reprise de la tâche affectée par la panne ne durera que R_i unités de temps en plus (au lieu de C_i si on devrait reprendre entièrement la tâche). Nous n'étudierons pas en détail cette approche dans le cadre de ce document.

P2 *On réxécute l'instance complète*. Ceci requiert davantage de temps processeur, mais en contrepartie, les sauvegardes de contexte ne seront plus requises qu'en fin de tâche. Cela limite ainsi fortement les surcoûts engendrés. Notons que ce cas peut être considéré comme un cas particulier du cas *P1*.

P3 *Enfin, la dernière option consiste à simplement abandonner l'instance en cours et à poursuivre l'exécution sur $m-1$ processeurs, en supposant que l'instance en cours de τ_{i_0} est terminée*. Ce scénario permet de ne pas sauvegarder les contextes des tâches, donc n'engendre pas du surcoût. Mais il nécessite de

déterminer auparavant si la perte d'une instance risque ou non d'avoir un impact sur le fonctionnement de l'application. Dans ce cas, il faut juste s'assurer que l'application peut se reconfigurer sans que ne se produisent de fautes temporelles.

4.1.3 Incidence sur les paramètres des tâches

Notre objectif est donc de permettre la reconfiguration de l'application de sorte à ce que la reprise éventuelle, totale ou partielle d'une tâche, ne provoque le dépassement d'échéance d'aucune autre tâche. En ce qui concerne la tâche reprise τi_0, le cas le pire se produit quand la panne a lieu alors que la tâche a presque fini son exécution, et que celle-ci se termine juste au moment de son échéance (voir figure 4.1). Dans ce cas, il faut

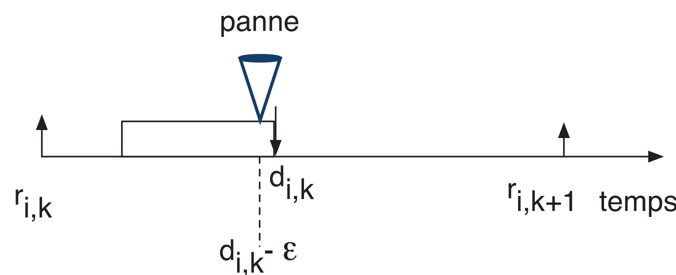


Figure 4.1 – Cas le pire d'occurrence d'une panne

ré-exécuter la tâche, et celle-ci ne peut plus respecter son échéance. Nous pouvons dans ce cas envisager deux scénarii :

H1 Chaque tâche τ_i dipose de deux délais critiques D_i et D'_i, de sorte que chaque instance de la tâches dispose de deux échéances : une échéance principale pour le fonctionnement normal et une échéance de reprise. Dans ce cas, nous devons avoir : $C_i \leq D_i \leq D_i + C_i \leq D'_i \leq T_i$ (voir la figure 4.2).

H2 Dans le cas où la tâche est totalement ré-exécutée, on peut également considérer une modification des paramètres de la tâche. Celle-ci est ré-exécutée à partir de l'instant t_p, qui est considéré comme sa nouvelle date d'activation (voir la figure 4.3). On effectue donc une translation de toutes les dates d'activation et de toutes les échéances à venir. Concrètement, ceci revient à supprimer la tâche τ_{i0} et à

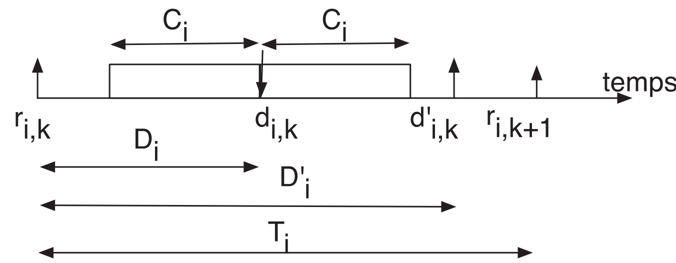


Figure 4.2 – Paramètres d'une tâche : échéance principale et échéance de reprise

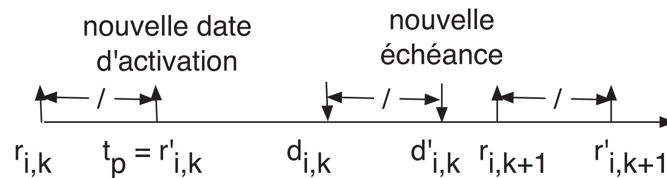


Figure 4.3 – Paramètres décalés d'une tâche dans l'optique d'une reprise totale

introduire une nouvelle tâche $\tau'_{i_0} =< t_p, C_{i_0}, D_{i_0}, T_{i_0} >$.

H3 Dans le cas où il y a ré-exécution partielle, une partie de la duré $C'_{i_0} \leq C_{i_0}$ doit être ré-exécutée, avant une date déchéance d'_{i_0}. Pour gérer cette reprise, nous proposons de considérer la partie à reprendre comme une tâche apériodique à contrainte stricte.

La reprise doit donc répondre aux règles suivantes :

C1 La tâche concernée doit pouvoir être exécutée en respectant les modalités choisies (i.e. en respectant sa nouvelle échéance si elle n'est pas abandonnée).

C2 Si une panne survient, aucune des tâches non affectées par la panne ne doit rater son échéance.

Dans les paragraphes qui suivent, nous présentons quelques résultats concernant la reprise sous l'une ou l'autre des hypothèses envisagées, lorque les tâches sont ordonnancées par EDF, puis par PF ou PD[2].

4.2 Quelques résultats concernant la reprise sous ordonnancement EDF

Notons tout d'abord que, dans les cas où les dates d'activation ne sont pas modifiées (le système reste à départs simultanés), il suffit de vérifier que toutes les échéances sont

respectées d'ici la fin de l'hyperpériode en cours (Ceci signifie que la période transitoire de reconfiguration doit être terminée à la fin de l'hyperpériode courante). En effet, si $kT \leq t_p < (k+1)T$, toutes les tâches sont réactivées à l'instant $(k+1)T$, et le système bascule sur le schéma à $m-1$ processeurs. Les figures 4.4 et 4.5 qui donnent les séquences d'ordonnancement *EDF* sur 2 puis 3 processeurs du système de tâches

$$\tau = \{(0,3,6,6);(0,2,6,6);(0,5,12,12);(0,5,12,12)\}$$

illustrent cela. Sans perte de généralité, nous supposons dans nos illustrations que $0 \leq t_p < T$.

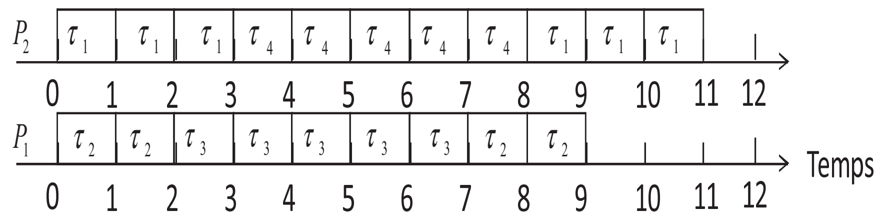


Figure 4.4 – Ordonnancement EDF d'un système de tâches sur 2 processeurs.
Le système de tâches ordonnancé est
$\tau = \{(0,3,6,6);(0,2,6,6);(0,5,12,12);(0,5,12,12)\}$

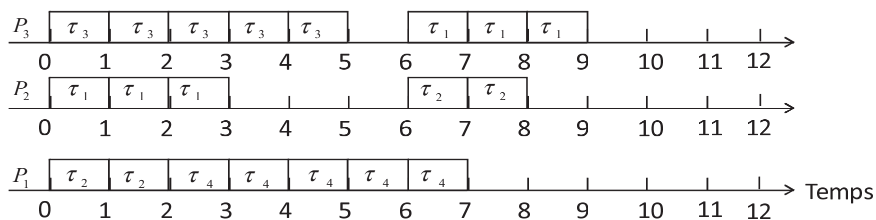


Figure 4.5 – Ordonnancement EDF d'un système de tâches sur 3 processeurs.
Le système de tâches ordonnancé est
$\tau = \{(0,3,6,6);(0,2,6,6);(0,5,12,12);(0,5,12,12)\}$

La figure 4.6 illustre un ordonnancement sur 3 processeurs du système de tâches

$$\tau = \{(0,3,6,6);(0,2,6,6);(0,5,12,12);(0,5,12,12)\}$$

avec une panne à l'instant $t_p = 9$ qui touche le processeur P_3. La tâche affectée par la panne est la tâche τ_1 et l'instant $t = 9$ correspond à la fin de son instance courante ($2^{ième}$ instance). La panne a lieu durant la première hyperpériode. On constate que sur la deuxième hyperpériode, la séquence correspond à un ordonnancement sur deux processeurs (conf. 4.4) du même système de tâches.

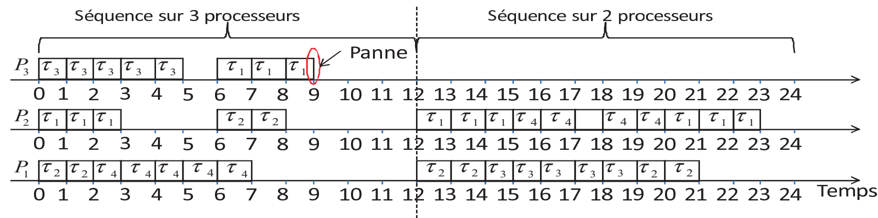


Figure 4.6 – Panne processeur en fin d'instance d'une tâche

4.2.1 Cas S1 et S2-P3

Nous envisageons tout d'abord les cas **S1** et **S2**-*P3* : soit aucune partie d'exécution n'est perdue, soit on abondonne l'exécution de l'instance en cours de la tâche touchée par la panne. Nous montrons que la réorganisation du système ne provoque aucune faute temporelle. Nous supposons, afin de rendre la stratégie d'ordonnancement complètement déterministe, que les ex-aequo sont départagés par le numéro de la tâche : en cas d'égalité des échéances, la tâche de plus petit numéro est élue.

Proposition 6. *Soit $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ un ensemble de tâches périodiques indépendantes, à départs simultanés, ordonnançable par EDF sur $m-1$ et sur m processeurs. Si une panne survient à l'instant t_p, et si l'instance en cours sur le processeur fautif est abandonnée, alors l'exécution pourra se poursuivre sur $m-1$ processeurs, sans faute temporelle.*

Preuve

Avant de donner la preuve du théorème nous précisons quelques notations que nous

utiliserons pour cette preuve. Soit τ un système de n tâches, nous notons O^m l'ordonnancement EDF de ce système de tâches sur m processeurs et O^{m-1} son ordonnancement EDF sur $m-1$ processeurs. La preuve est basée sur l'analyse de la charge processeur. Nous allons montrer qu'une tâche à un instant donné ne peut pas avoir exécuté plus de charge processeur sur $m-1$ que sur m processeurs. Dans ce cas, l'indisponibilité d'un processeur à un instant donné ne peut pas provoquer de dépassement d'échéance.
Nous rappelons que :

- $RCT_i(t,O^m)$ désigne la charge processeur restante à l'instance courante de la tâche τ_i à l'instant t pour terminer son exécution selon l'ordonnancement O^m ;
- $\overline{RCT}_i(t,O^m)$ désigne le temps processeur déjà reçu par l'instance courante de la tâche τ_i à l'instant t selon l'ordonnancement O^m ;
- $Ctx(t,O^m)$ est le contexte de l'ordonnancement O^m à l'instant t, c'est à dire le couple $(ListExec, ListAtt)$. $Ctx.ListExec$ est la liste des tâches en cours d'exécution à l'instant t et $Ctx.ListAtt$ est la liste des tâches en attente à cet instant.

Lemme 5. *1.* $\forall \tau_i \in \tau, \forall t, RCT_i(t,O^m) \leq RCT_i(t,O^{m-1})$.

2. $\forall t, \tau_i \in Ctx(t,O^{m-1}).ListExec \Longrightarrow \tau_i \in Ctx(t,O^m).ListExec$ *ou* $RCT_i(t,O^m) = 0$

Remarque 1. *Notons pour commencer que si* $\tau_i \in Ctx(t,O^m).ListExec$ *et* $\tau_j \in Ctx(t,O^m).ListAtt$*, alors* $d_i < d_j$ *ou* $d_i = d_j$ *et* $i < j$.

Preuve du lemme

Nous prouvons le lemme par induction sur t.
Supposons $t = 0$. Alors,
1) $RCT_i(0,O^m) = RCT_i(0,O^{m-1}) = C_i \;\; \forall \tau_i$ puisqu'aucune tâche n'a commencé son exécution à $t = 0$.
Supposons maintenant $Ctx(0,O^m).ListExec = \{\tau_{i_1},...,\tau_{i_m}\}$ et
$Ctx(0,O^{m-1}).ListExec = \left\{\tau_{j_1},...,\tau_{j_{m-1}}\right\}$.
Alors $(d_{i_1} \leq d_{i_2} \leq \ldots \leq d_{i_m}$ et $d_{i_k} = d_{i_{k+1}} \Rightarrow i_k < i_{k+1})$ et si $\tau_{i_k} \notin \{\tau_{i_1},...,\tau_{i_m}\}$ alors $d_{i_m} < d_{i_k}$ ou $(d_{i_m} = d_{i_k}$ *et* $i_m < i_k)$ puis
$(d_{j_1} \leq d_{j_2} \leq \ldots \leq d_{j_{m-1}}$ et $d_{j_k} = d_{j_{k+1}} \Rightarrow j_k < j_{k+1})$ et si $\tau_{j_k} \notin \left\{\tau_{j_1},...,\tau_{j_{m-1}}\right\}$ alors $d_{j_{m-1}} < d_{j_k}$ ou $(d_{j_{m-1}} = d_{j_k}$ *et* $j_{m-1} < j_k)$.

Donc $i_1 = j_1, ..., i_{m-1} = j_{m-1}$.

Par suite, si $\tau_i \in Ctx(0, O^{m-1})$, cela veut dire que $i \in \{j_1, ..., j_{m-1}\}$; donc $i \in \{i_1, ..., i_m\}$ et finalement $\tau_i \in Ctx(0, O^m).ListExec$. Et donc,

2) $\tau_i \in Ctx\left(0, O^{m-1}\right) \Rightarrow \tau_i \in Ctx\left(0, O^m\right).ListExec$.

Supposons maintenant les points 1) et 2) vérifiés pour t.

Considérons l'instant $t+1$.

Étant donnée une tâche τ_i, par hypothèse de récurrence nous avons :

$$\begin{cases} RCT_i(t, O^m) \leq RCT_i(t, O^{m-1}) \\ \tau_i \in Ctx(t, O^{m-1}).ListExec \Rightarrow \tau_i \in Ctx(t, O^m).ListExec \ ou \ RCT_i(t, O^m) = 0. \end{cases}$$

1)

a- Si τ_i a été activée à $t+1$, alors $RCT_i(t+1, O^m) = RCT_i(t+1, O^{m-1}) = C_i$

b- Sinon :

- si $RCT_i(t, O^m) = 0$ alors $RCT_i(t+1, O^m) = 0 \leq RCT_i(t+1, O^{m-1})$,

- si $RCT_i(t, O^m) > 0$ alors :

* si $\tau_i \in Ctx(t, O^m).ListExec$ alors $RCT_i(t+1, O^m) = RCT_i(t, O^m) - 1 \leq RCT_i(t, O^{m-1}) - 1 \leq RCT_i(t+1, O^{m-1})$;

* si $\tau_i \notin Ctx(t, O^m).ListExec$ alors $\tau_i \notin Ctx(t, O^{m-1}).ListExec$ et par suite $RCT_i(t+1, O^m) = RCT_i(t, O^m)$ et $RCT_i(t+1, O^{m-1}) = RCT_i(t, O^{m-1})$ donc $RCT_i(t+1, O^m) \leq RCT_i(t+1, O^{m-1})$.

2)

Soit $\tau_i \in Ctx(t+1, O^{m-1}).ListExec$. Supposons que $Ctx(t+1, O^{m-1}).ListExec = \left\{\tau_{i_1}, ..., \tau_{i_q}\right\}$ avec $q \leq m-1$. Nous avons $d_{i_1} \leq \ldots \leq d_{i_q}$ ou $d_{i_k} = d_{i_{k+1}} \Rightarrow i_k < i_{k+1}$.

a- Si $q < m-1$

Comme $\tau_i \in Ctx(t+1, O^{m-1}).ListExec$, nous avons $RCT_i(t+1, O^{m-1}) > 0$. D'après le point 1 du lemme, nous avons

$$\left\{j / RCT_j(t+1, O^m) > 0\right\} \subset \left\{j / RCT_j(t+1, O^{m-1}) > 0\right\} \ (3)$$

et donc,

$$\left|\left\{j/RCT_j(t+1,O^{m-1})>0\right\}\right|<m-1 \Rightarrow \left|\left\{j/RCT_j(t+1,O^{m})>0\right\}\right|<m-1.$$

Par conséquent toutes les tâches prêtes sont exécutées, que l'on ait m ou $m-1$ processeurs. Donc si $RCT_i(t+1,O^m)>0$ alors $\tau_i \in Ctx(t+1,O^m).ListExec$.

b- Si $q=m-1$

Considérons $\tau_i \in Ctx(t+1,O^{m-1}).ListExec$. Si $RCT_i(t+1,O^m)$ >0, nous avons $\left|\left\{j/RCT_j(t+1,O^{m-1})>0 \; et \; (d_j<d_i \; ou \; (d_j=d_i \; et \; j<i))\right\}\right|<m-1$, donc d'après (3) $\left|\left\{j/RCT_j(t+1,O^{m})>0 \; et \; (d_j<d_i \; ou \; (d_j=d_i \; et \; j<i))\right\}\right|<m-1$.

Par suite, il y a au plus $m-2$ tâches prêtes plus prioritaires que τ_i et m processeurs. Donc τ_i est ordonnancée, c'est à dire que $\tau_i \in Ctx(t+1,O^m).ListExec$. Cela termine la preuve du lemme.□

Preuve du théorème

A partir du lemme, nous pouvons conclure que sur une architecture de m processeurs, si un processeur tombe en panne à un instant t_p, l'exécution de toutes les tâches est au moins aussi avancée que si elles s'exécutaient sur une plateforme de $m-1$ processeurs. Les tâches pourront donc terminer leur instance sans rater leur échéance. □

Nous pouvons déduire de la preuve le corollaire suivant qui montre qu'il suffira de vérifier que le système est ordonnançable sur $m-1$ processeurs ; l'hypothèse d'ordonnançabilité sur m processeurs n'est pas nécessaire.

Corollaire 2. *Si un système de tâches indépendantes et à départs simultanés est ordonnançable par EDF sur $m-1$ processeurs, il l'est sur m processeurs avec $m \in \mathbb{N}$ et $m \geq 2$.*

Malheureusement, si les tâches ne sont pas indépendantes, le lemme n'est plus valide comme l'illustre l'exemple suivant : considérons le système de 3 tâches

$$\tau=\{(0,1+1(R),4,4);(0,4(R),8,8);(0,3,4,4)\}$$

Les tâches τ_1 et τ_2 partagent la ressource critique R. La tâche τ_1 utilise la ressource R

pendant une unité de temps après s'être exécutée une unité de temps. La tâche τ_2 utilise la ressource pendant les 4 unités de temps durant lesquelles elle s'exécute. Ce système de tâches est ordonnançable par *EDF* sur deux processeurs (voir la figure 4.7) mais ne l'est plus quand on passe sur une architecture composée de trois processeurs (voir la figure 4.8). En effet, les 3 tâches commencent leur première instance à $t = 0$ puisqu'il y a 3 tâches et 3 processeurs libres. L'unique instance de la tâche τ_2 sur l'hyperpériode commence son exécution à l'instant 0 et prend donc la ressource R avant τ_1 qui ne la demande qu'à l'instant $t = 1$. τ_1 est mise en attente de la ressource critique R et ne l'aura qu'à l'instant $t = 4$ après que τ_2 ait fini son instance. L'exécution de la première instance de τ_1 dépasse donc son échéance comme l'illustre la figure 4.8.

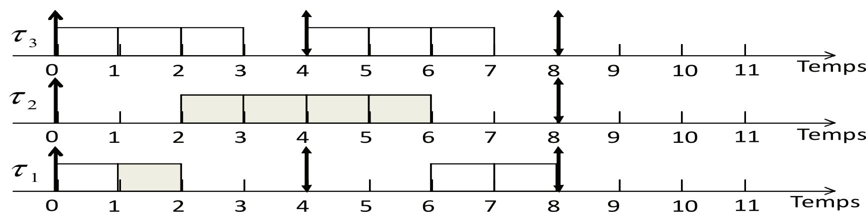


Figure 4.7 – Ordonnancement EDF d'un système de tâches sur 2 processeurs avec partage d'une ressource critique.
le système de tâches ordonnancé est $\tau = \{(0,1+1(R),4,4);(0,4(R),8,8);(0,3,4,4)\}$

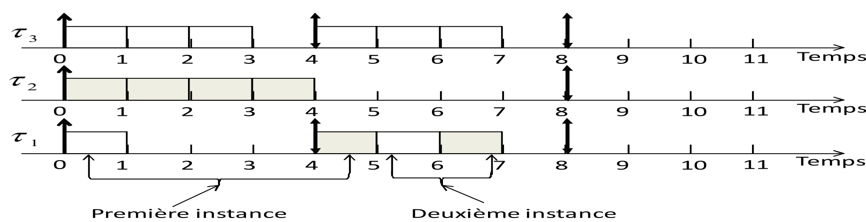


Figure 4.8 – Ordonnancement EDF d'un système de tâches sur 3 processeurs avec partage d'une ressource critique.
le système de tâches ordonnancé est $\tau = \{(0,1+1(R),4,4);(0,4(R),8,8);(0,3,4,4)\}$

4.2.2 Cas S2 - P1 et P2

Nous supposons maintenant qu'au moins une partie de la tâche doit être reprise. Nous envisageons tout d'abord le schéma H1 : les tâches ont deux échéances. Ce schéma ne semble cependant pas prometteur, car même si la portion à reprendre est très réduite, nous ne pouvons pas conclure de façon systématique. Nous avons envisagé le schéma où $C_i \leq D_i \leq \frac{P_i}{2}$, avec comme second délai critique la période. Considérons le système $\tau = \{(0,4,4,8),(0,4,8,16),(0,4,8,16),(0,2,4,8),(0,2,4,8)\}$. Ce système est ordonançable sur 2 et sur 3 processeurs (voir les figures 4.9 et 4.10). Cependant, si on considère trois processeurs, et si on suppose que le processeur 1 tombe en panne à l'instant $t_p = 3$, la première isntance de la tâche τ_1 doit être reprise, mais ceci entraîne un dépassement d'échéances des tâches τ_3 et τ_5 respectivement aux instants $t = 8$ et $t = 12$ (voir figure 4.11).

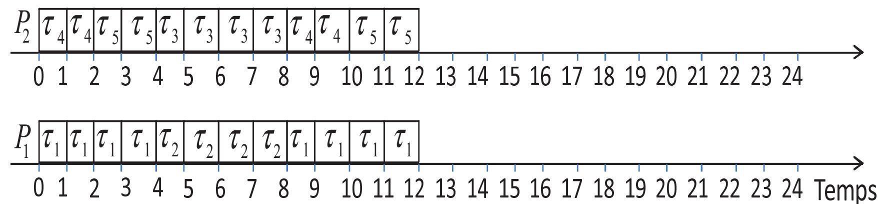


Figure 4.9 – Ordonnancement EDF d'un système de tâches sur 2 processeurs.
Le système de tâches ordonnancé est
$\tau\{(0,4,4,8),(0,4,8,16),(0,4,8,16),(0,2,4,8),(0,2,4,8)\}$

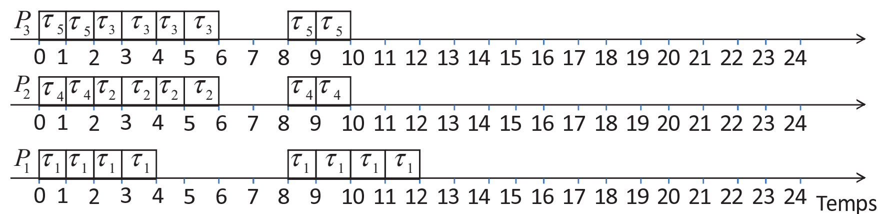


Figure 4.10 – Ordonnancement EDF d'un système de tâches sur 3 processeurs.
Le système de tâches ordonnancé est
$\tau\{(0,4,4,8),(0,4,8,16),(0,4,8,16),(0,2,4,8),(0,2,4,8)\}$

Et si nous envisageons le schéma H2, il y a également dépassement d'échéance, cette

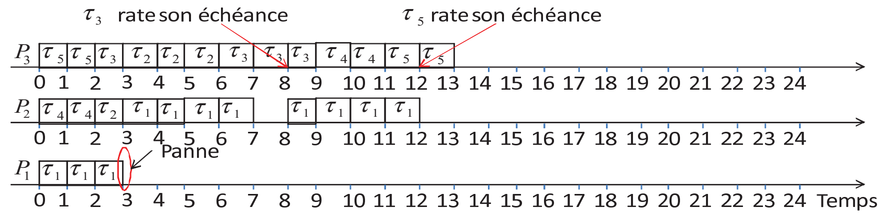


Figure 4.11 – Ordonnancement EDF d'un système de tâches sur 3 processeurs avec une panne d'un processeur à l'instant $t_p = 3$.
Le système de tâches ordonnancé est
$\tau\{(0,4,4,8),(0,4,8,16),(0,4,8,16),(0,2,4,8),(0,2,4,8)\}$

fois de la tâche τ_3 à $t = 8$. En effet, la tâche τ_1 est réactivée avec l'instant t_p comme sa date de première activation (voir figure 4.12), ce qui décale ses échéances. Les paramètres temporels de la tâche τ_1 deviennent donc $(3,4,4,8)$. La reconfiguration de τ_1 a répoussé l'exécution de la tâche τ_3 l'amenant ainsi à rater son échéance à l'instant $t = 8$.

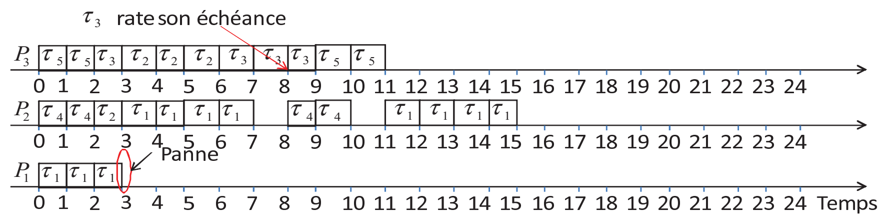


Figure 4.12 – Ordonnancement EDF d'un système de tâches sur 3 processeurs avec une panne d'un processeur à l'instant $t_p = 3$.
Le système de tâches orrdonnancé est
$\tau = \{(0,4,4,8),(0,4,8,16),(0,4,8,16),(0,2,4,8),(0,2,4,8)\}$

Ces observations nous montre que la gestion des pannes quand on ordonnance le système avec *EDF* pose très vite des problèmes. Pour cette raison, nous nous sommes plutôt orientés vers les ordonnancements équitables, puisque ce sont ceux qui ont la plus grande puissance d'ordonnancement.

4.3 Quelques résultats concernant la reprise sous ordonnancements équitables

Nous considérons maintenant des sytèmes de tâches ordonnancées par une stratégie équitable (PF ou PD^2).

4.3.1 Cas S1 et S2-P3

Nous ne nous attardons pas sur cette partie car nous sommes dans un contexte où les tâches du système sont à départs simultanés et à échéances sur requêtes et en plus le facteur d'utilisation U est tel que $U \leq m-1$. Cela veut donc dire que le système est ordonnançable sur m et $m-1$ processeurs. De plus, la proposition 6 est vérifiée pour un ordonnacement P-équitable car au plus $m-1$ tâches sont urgentes à chaque instant du fait des propriétés de la politique d'ordonnancement et par conséquent la propriété

$$\begin{cases} RCT_i(t,O^m) \leq RCT_i(t,O^{m-1}) \\ \tau_i \in Ctx(t,O^{m-1}).ListExec \Rightarrow \tau_i \in Ctx(t,O^m).ListExec\ ou\ RCT_i(t,O^m) = 0. \end{cases}$$

restera vérifiée dans ce contexte.

4.3.2 S2 - P1 et P2

Une partie ou la totalité de la tâche doit être reprise. Nous envisageons l'hypothèse *H1*, c'est à dire que les tâches ont deux échéances. Pour une tâche $\tau_i \in \tau$ donnée, sa deuxième échéance doit être telle que la tâche puisse avoir au moins C_i unités de temps pour sa reprise. Nous devons donc avoir $T_i - T_i^{'} \geq C_i$ d'où $T_i^{'} \leq T_i - C_i$.
Supposons que $\forall \tau_i \in \tau, \tau_i = (0, C_i, T_i - C_i, T_i, T_i)$. Dans ce cas, à $T_i - C_i$ unités de temps après l'activation de la tâche τ_i, elle aura effectué au moins $\left\lceil \frac{C_i}{T_i - C_i} * (T_i - C_i) \right\rceil = C_i$ unités de temps processeur (nous considérons la première instance mais cela est valable sur toutes les instances). Mais une telle hypothèse suppose que nous donnons un sens à l'équité si les tâches sont à échéances contraintes et surtout, nous devons garantir l'ordonnançabilité sous cette hypothèse, c'est à dire avoir un critère d'ordonnançabilité pour les systèmes de tâches à échéances contraintes. De plus il faut montrer que la bascule

d'un mode à l'autre se fait bien, ce qui n'est pas évident.

Si nous envisageons le schéma *H2*, cela nécessite que l'on sache déterminer si un système de tâches à départs différés est ou non ordonnançable dans un contexte d'ordonnancement P-équitable.

Et enfin, nous envisageons le schéma *H3*. Cela nécessite de savoir comment intégrer des tâches apériodiques à contraintes strictes dans un contexte d'ordonnancement P-équitable. Mais en amont, pour être sûr que la tâche reprise aura assez de temps processeur pour cela, il faut revenir sur la notion d'équité dans un contexte où il y a des échéances strictes. On a besoin d'ordonnancer des systèmes tâches définis de la sorte : $\tau = \{\tau_i(0, C_i, T_i - C_i, T_i), i = 1 \ldots n \in \mathbb{N}\}$. Le chapitre suivant donne un cadre à l'ordonnancement équitable de ce type de tâches.

Conclusion du chapitre

Dans ce chapitre nous avons proposé plusieurs techniques de reconfiguration en cas de panne processeur. Ces techniques, pour être mises en œuvre nécessitent la mise en place de résultats concernant d'une part l'extension de la notion d'ordonnancement P-équitable aux systèmes de tâches à départs différés et à échéances contraintes, et d'autre part l'intégration de tâches apériodiques dans le système.
Ces résultats font l'objet des chapitres suivants. L'étude de la mise en œuvre effective des techniques de reconfiguration devra ensuite être effectuée. Dans cette étude, nous n'avons envisagé que la panne d'un seul processeur et les systèmes considérés comportent des tâches indépendantes. Des travaux futurs élargiront le contexte de mise en œuvre de notre solution.

CHAPITRE 5

ETUDE DES ORDONNANCEMENTS P-ÉQUITABLES POUR LES TÂCHES À DÉPARTS DIFFÉRÉS ET À ÉCHÉANCES CONTRAINTES

Les schémas de reconfiguration envisagés dans le chapitre précédent nécessitent de considérer la P-équité pour des systèmes de tâches à départs différés et à échéances contraintes.
Notre objectif ici est d'étendre la définition de la P-équité à ce contexte et de dégager une condition suffisante d'ordonnançabilité. Le problème de l'ordonnancement des tâches à échéances contraintes sous ordonnancement P-équitable a été abordé par Ramamurthy dans [98]. Mais ces travaux se sont limités au cadre des tâches à départs simultanés. A notre connaissance, c'est la première fois qu'on considère simultanément les tâches à départs différés et à échéances contraintes.
Au cours des simulations que nous avons menées, il est apparu que les systèmes de tâches à départs simultanés et à échéances contraintes respectant la condition $CH = \sum_{\tau_i \in \tau} \frac{C_i}{D_i} \leq m$ étaient ordonnançables sur m processeurs par PF ainsi que les systèmes à départs différés à échéances sur requêtes respectant la condition $U \leq m$. Cela nous a conduit à énoncer le théorème suivant dont nous donnerons la preuve dans les sections à venir :

Theorème 5. *Tout système τ de tâches périodiques indépendantes vérifiant $\sum_{\tau_i \in \tau} \frac{C_i}{D_i} \leq m$ a un ordonnancement P-équitable sur une architecture composée de m processeurs.*

5.1 Motivation et état de l'art

L'un des mécanisme de reprise après panne suppose la modification de la date d'activation de la tâche affectée par la panne. La tâche ainsi modifiée sera traitée comme une tâche à départ différé. Par ailleurs, pour préserver du temps avant la réactivation d'une tâche τ_i, une des approches consiste à avoir un délai absolu D_i vérifiant $D_i + C_i \leq T_i$. Dans ce cas, les tâches auront des délais absolus plus petits que les période et par conséquent ce sont des tâches à échéances contraintes. Il s'en suit alors le besoin d'étudier

l'ordonnançabilité sous un ordonnancement P-équitable dans ces contextes.
Les premiers algorithmes d'ordonnancement P-équitables étaient restreints aux tâches périodiques indépendantes à départs simultanés et à échéances sur requêtes. C'est dans ce contexte que Baruah et al dans [18] ont établi l'optimalité des ordonnancements P-équitables et ont donné une condition nécessaire et suffisante d'ordonnançabilité.
Les premières extensions du contexte de mise en œuvre de ces ordonnancements ont été apportées par Srinivasan et Anderson qui se sont intéressés au problème en proposant dans [5, 106] d'autres modèles de tâches plus flexibles du point de vue de l'ordonnancement. Ils ont proposé les modèles de tâches Early-Release (ER), Instra-Sporadic (IS) et Instra-Sporadic Généralisé (GIS). Cependant ces extensions ne répondent pas à nos besoins car notre objectif est d'alléger les hypothèses sur le modèle de tâche ou le modèle d'exécution des tâches en considérant simplement le modèle de Liu et Layland. D'où la nécessité de nous intéresser à l'extension du contexte d'utilisation des ordonnancements P-équitables.

5.2 Condition suffisante d'ordonnançabilité

Dans cette section, nous allons donner une preuve du théorème 5. Dans le cas général, nous ne disposons plus d'une condition nécessaire et suffisante d'ordonnançabilité. Nous n'avons qu'une condition suffisante.

5.2.1 Formulation de l'équité dans ce contexte

Dans un premier temps, nous revenons sur la définition de la *P-équité*. Elle a été définie dans le contexte des *départs simultanés* et des *échéances sur requêtes*. Nous l'adaptons au contexte des *départs différés* et des *échéances contraintes*. Un ordonnancement est équitable si toute tâche s'exécute régulièrement dans chacune de ses périodes d'activité (entre activation et échéance) et est inactive entre chaque échéance et l'activation qui suit. Formellement, lors d'un ordonnancement équitable, une tâche τ_i donnée devrait avoir reçu à chaque instant t, $W_i^{ideal}(0,t)$, où la fonction $W_i^{ideal}(0,t)$ est définie

par :

$$W_i^{ideal}(0,t) = \begin{cases} 0 \; si \; t \in [0, r_i) \\ \\ k * C_i + \frac{C_i}{D_i} * (t - k * T_i - r_i) \; si \\ t \in [k * T_i + r_i, k * T_i + D_i + r_i) \\ \\ (k+1) * C_i \\ si \; t \in [k * T_i + D_i + r_i, (k+1) * T_i + r_i). \end{cases}$$

$k = \left\lfloor \frac{t}{T_i} \right\rfloor$ représente le numéro de l'instance en cours.

Définition 18. *Un ordonnancement O est P-équitable si et seulement si*

$$\forall \tau_i \in \tau, \forall t \in \mathbb{N} :: -1 < retard(O, \tau_i, t) < 1 \;\; avec$$

$$retard(O, \tau_i, t) = W_i^{ideal}(0,t) - W_i^{O}(0,t) = W_i^{ideal}(0,t) - \sum_{j=0}^{j=t-1} O_i(j).$$

Nous redéfinissons également dans ce contexte la notion de sous-tâche. Les fenêtres d'exécution des sous-tâches d'une tâche sont incluses dans les périodes d'activité de la tâche : la $j^{ième}$ $(j \geq 1)$ sous tâche τ_i^j appartient à la $k^{ième}$ $(k \geq 1)$ instance de la tâche τ_i avec $k = \left\lceil \frac{j}{C_i} \right\rceil$ et $I_i^j \subset [r_i + (k-1) * T_i, r_i + (k-1) * T_i + D_i)$. La fenêtre I_i^j est de la forme $\left[r_i^j, d_i^j\right)$ avec $r_i^j = r_i + kT_i + \left\lfloor \frac{j-kC_i}{CH_i} \right\rfloor$ et $d_i^j = r_i + kT_i + \left\lceil \frac{j+1-kC_i}{CH_i} \right\rceil$.
L'exemple de la figure 5.1 illustre une exécution P-équitable de la tâche $\tau_i = (2,3,5,8)$.

5.2.2 Preuve du théorème

La preuve que nous proposons pour le théorème est une adaptation de la preuve proposée par Baruah et al. dans [18] pour démontrer le théorème 1. La preuve s'appuie sur la théorie des graphes. Nous montrons qu'un ordonnancement P-équitable existe pour le système de tâches sur toute fenêtre temporelle $[0, L)$. Dans la suite, nous notons $DT_i(t)$ la demande processeur totale de toutes les sous tâches de la tâche τ_i dont la fenêtre d'exécution est dans l'intervalle $[0, t]$. Donc, $DT_i(t) = j$, où j est tel que

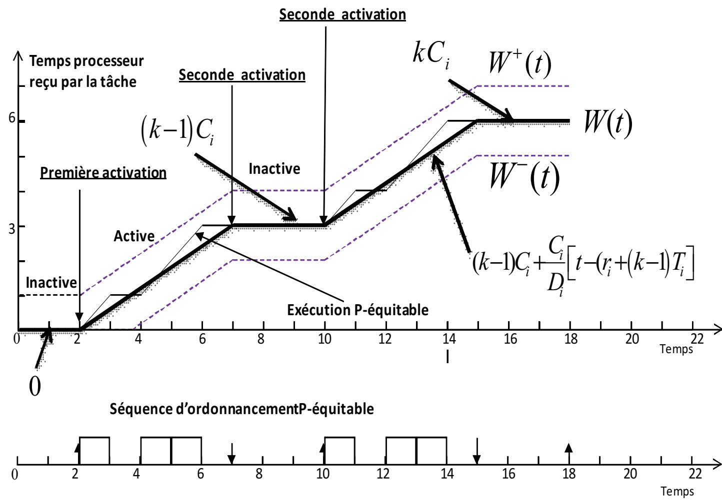


Figure 5.1 – Ordonnancement P-équitable de la tâche $\tau_i = (2,3,5,8)$

$r_i^j \leq t < r_i^{j+1}$.

Tout d'abord, nous construisons un graphe valué $G(L)$ puis nous prouvons que si ce graphe a un flot entier de capacité $\sum_{\tau_i\in\tau} DT_i(L)$ alors, il existe un ordonnancement Pfair pour τ.

Définition 19. *Le graphe $G(L)$ est un graphe orienté valué défini par : $G(L) = (V,E)$ avec, $V = V_0 \cup V_1 \cup V_2 \cup V_3 \cup V_4 \cup V_5$ et $E = E_0 \cup E_1 \cup E_2 \cup E_3 \cup E_4$ où*

- $V_0 = \{< source >\}$
- $V_1 = \{< 1, \tau_i >, \tau_i \in \Gamma\}$
- $V_2 = \{< 2, \tau_{i,0}, 0 >, i = 1..n \ t.q.\ r_i > 0\} \cup$ $\{< 2, \tau_i^j >, (i, j)\ t.q.\ i = 1..n, j \in [0, DT_{\tau_i}(L))\} \cup \{< 2, \tau_{i,0}, j >, (i, j)\ t.q.\ i = 1..n, j \in [1, \left\lfloor \frac{L-r_i}{T_i} \right\rfloor]\}$
- $V_3 = \{< 3, \tau_{i,0}, t >, (i,t)\ t.q.\ i = 1..n, t \in [0, r_i)\} \cup$ $\{< 3, \tau_i, t >, (i,t)\ t.q.\ i = 1..n, t \in [k.T_i + r_i, k.T_i + r_i + D_i),\ avec\ 1 \leq k \leq \left\lfloor \frac{L-r_i}{T_i} \right\rfloor\} \cup$ $\{< 3, \tau_{i,0}, t >, (i,t)\ t.q.\ i = 1..n, t \in [k.T_i + D_i, (k+1).T_i)\ avec\ 1 \leq k \leq \left\lfloor \frac{L-r_i}{T_i} \right\rfloor\}$
- $V_4 = \{< 4, t >, t \in [0, L)\}$
- $V_5 = \{< puit >\}$.

Les arcs ainsi que leur valuation sont définis par :

- $E_0 = \{(< source >, < 1, \tau_i >, DT_{\tau_i}(L)), i = 1..n\}$

- $E_1 = \{(< 1, \tau_i >, < 2, \tau_{i,0}, 0 >, 0), i = 1..n\ t.q.\ r_i > 0\} \cup$
  $\{(< 1, \tau_i >, < 2, \tau_i^j >, 1), j \in [0, DT_{\tau_i}(L))\} \cup$
  $\{(< 1, \tau_i >, < 2, \tau_{i,0}, j >, 0), (i, j)\ t.q.\ i = 1..n, j \in \left[1, \left\lfloor \frac{L-r_i}{T_i} \right\rfloor\right]\}$
- $E_2 = \{(< 2, \tau_{i,0}, 0 >, < 3, \tau_{i,0}, t >, 0), (i, t)\ t.q.\ i = 1..n, t \in [0, r_i)\} \cup$
  $\{(< 2, \tau_i^j >, < 3, \tau_i, t >, 1), (i, j, t)\ t.q.\ i = 1..nj \in [0, DT_{\tau_i}(L)), t \in [r_i^j, d_i^j)\} \cup$
  $\{(< 2, \tau_{i,0}, j >, < 3, \tau_i, t >, 0)(i, j, t)\ t.q.\ i = 1..n,\ j \in [1, \left\lceil \frac{L-r_i}{T_i} \right\rceil), t \in [k.T_i + r_i + D_i, (k+1).T_i + r_i)\ avec\ 1 \leq k \leq \left\lfloor \frac{L-r_i}{T_i} \right\rfloor]t\}$
- $E_3 = \{(< 3, \tau_{i,0}, t >, < 4, t >, 0), t \in [0, r_i)\} \cup$
  $\{(< 3, \tau_i, t >, < 4, t >, 1), t \in [k.T_i + r_i, k.T_i + r_i + D_i)\ avec\ 1 \leq k \leq \left\lfloor \frac{L-r_i}{T_i} \right\rfloor\} \cup$
  $\{(< 3, \tau_{i,0}, t >, < 4, t >, 0), (i, t)\ t.q.\ i = 1..n, t \in [k.T_i + r_i + D_i, (k+1).T_i)\ avec\ 1 \leq k \leq \left\lfloor \frac{L-r_i}{T_i} \right\rfloor\}.$
- $E_4 = \{(< 4, t >, < puit >, m), t \in [0, L)\}.$

Intuitivement, la construction du graphe $G(L)$ s'interprète de la façon suivante :
Chaque nœud de V_1 représente une tâche τ_i, et la capacité de l'arc qui relie V_0 au nœud de V_1 représentant la tâche τ_i est la demande processeur de la tâche τ_i dans l'intervalle de temps $[0, L)$.
Chaque nœud de V_2 représente soit une sous tâche d'une tâche, soit une période d'inactivité d'une tâche. L'inactivité d'une tâche τ_i est due soit au fait que la tâche n'est pas encore activée, soit au fait qu'elle a atteint son échéance. Chaque nœud de V_2 est relié au nœud de V_1 qui correspond à sa tâche mère et la capacité de l'arc est égale à 1.
Chaque nœud de V_3 représente un instant d'exécution potentiel d'une sous tâche ou un instant d'inactivité d'une tâche. Chaque nœud de V_3 est relié à un nœud de V_2. Soit un nœud quelconque de V_3. S'il correspond à un temps d'exécution potentiel d'une sous tâche, c'est à dire un instant t qui appartient à sa fenêtre d'exécution, il est alors relié au nœud de V_2 correspondant à cette sous tâche et l'arc les reliant est de capacité 1. S'il correspond à un temps d'inactivité d'une tâche, alors il est relié au nœud de V_2 qui correspond à la période d'inactivité concernée et l'arc correspondant est de capacité 0.

Enfin, chaque nœud de V_4 correspond à un instant de la fenêtre temporelle $[0,L)$ et est relié aux noeuds de V_3 correspondant au même instant. Les arcs qui relient le noeud de V_4 aux noeuds de V_3 ont une capacité de 1.
Considérons, pour illustrer cette construction, une configuration τ de deux tâches à départs simultanés et à échéances contraintes, $\tau = \{u(0,3,5,8), v(0,2,3,4)\}$. L'intervalle d'étude pour ce système est égal à $[0,8)$. La figure 5.2 donne le graphe $G(8)$.

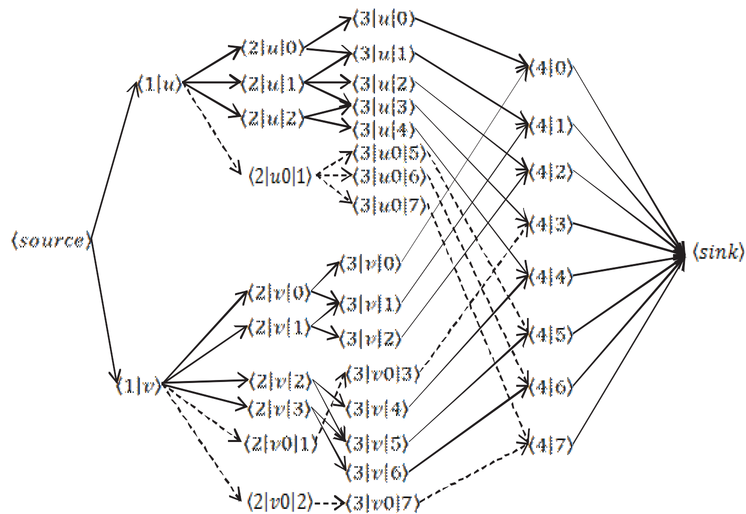


Figure 5.2 – Exemple de graphe :
Graphe valué $G(8)$, modélisant le système de tâche $\tau = \{u(0,3,5,8), v(0,2,3,4)\}$

Afin de prouver le théorème 5, nous énonçons d'abord le lemme suivant :

Lemme 6. *Si le graphe $G(L)$ a un flot entier de valeur $\sum_{\tau_i \in \tau} DT_i(L)$, alors il existe un ordonnancement P-équitable pour le système de tâche τ dans l'intervalle $[0,L)$.*

Preuve du lemme
Supposons que le flot entier existe. Nous définissons un ordonnancement O déduit du graphe $G(L)$ par :

Définition 20. *Soit f un flot entier du graphe $G(L)$ de valeur $\sum_{\tau_i \in \tau} DT_i(L)$. Nous défi-*

nissons O de la façon suivante : $\forall \tau_i \in \tau, \forall t \in \mathbb{N}$,

$$O_i(t) = \begin{cases} 1 \; si \; t \in [0,L) \wedge (\exists j \in [0,DT_i(L))) :: \\ \quad f((<2,\tau_i^j>,<3,\tau_i,t>,1)) = 1 \\ \\ 0 \qquad\qquad sinon \end{cases}$$

Nous allons montrer que l'ordonnancement O sur l'intervalle de temps $[0,L)$ est P-équitable. Comme la valeur du flot est égale à $\sum_{\tau_i \in \tau} DT_i(L)$, le flot entrant sur l'ensemble des noeuds de V_1 est maximal. Cela veut dire que le flot le long de chaque arc reliant la source $< source >$ à chaque noeud $< 1, \tau_i >$ vaut $DT_i(L)$. Chaque nœud de V_1 a exactement $DT_i(L)$ arcs sortants de capacité 1 et les autres arcs sortants ont une capacité nulle donc ils reçoivent un flot nul. Donc chaque nœud $< 2, \tau_i^j >$ de V_2 reçoit un flot entrant égal à 1. Il s'en suit alors que le nœud $< 2, \tau_i^j >$ a un arc sortant transportant un flot de valeur 1, et les autres arcs transportent un flot de valeur nulle. Chaque nœud de V_3 a un arc sortant. Cet arc transporte un flot égal au flot transporté par l'arc entrant. De même, chaque nœud de V_4 a un arc sortant qui transporte le même flot que celui transporté par l'ensemble des arcs entrants. Comme la capacité de l'arc sortant du nœud de V_4 est m (le nombre de processeurs), alors au plus m arcs entrants dans le nœud transportent un flot de valeur 1. Cela veut dire qu'il y a au plus m sous tâches ordonnancées. Cependant, potentiellement, il est possible qu'une tâche τ_i soit ordonnancée deux fois un même instant t (si $d_i^{j-1} = r_i^j$). Mais comme l'arc qui relie les nœuds $< 3, \tau_i, t >$ à $< 4, t >$ est de capacité 1, alors cette situation ne peut pas se produire : si cet arc transporte un flot de valeur 1, alors un seul arc entrant $< 3, \tau_i, t >$ transporte un flot de valeur non nulle. Par suite, deux sous tâches ordonnancées ne peuvent pas appartenir à la même tâche. Donc à chaque instant t, il existe au plus m sous tâches telles que $O_i(t) = 1$. De plus, $f((<2,\tau_i,j>,<3,\tau_i,t>,1)) = 1$ implique que $r_i^j \leq t < d_i^j$. Par suite, chaque sous tâche est bien ordonnancée dans sa fenêtre d'exécution. Enfin, on peut affirmer que toutes les sous tâches sont bien effectivement ordonnancées dans l'intervalle de temps $[0,L)$. En effet, il y a dans cet intervalle $\sum_{\tau_i \in \tau} DT_i(L)$ sous tâches. D'après la définition de O, le

nombre de sous tâches ordonnancées est égal à la valeur du flot entrant dans l'ensemble des nœuds de V_3. Par définition du flot, cette valeur est constante et vaut $\sum_{\tau_i \in \tau} DT_i(L)$. Il s'en suit donc que chaque sous tâche a bien été ordonnancée. L'ordonnancement O est donc P-équitable □

Nous allons maintenant prouver l'existence du flot entier de valeur $\sum_{\tau_i \in \tau} DT_i(L)$.

Définition 21. *Soit f un flot défini par :*

- $f((< source >, < 1, \tau_i >, DT_i(L))) = DT_i(L)$
- $\triangleright\ f((< 1, \tau_i >, < 2, \tau_{i,0}, 0 >, 0)) = 0$
  $\triangleright\ f((< 1, \tau_i >, < 2, \tau_i, j >, 1)) = 1$
  $\triangleright\ f((< 1, \tau_i >, < 2, \tau_{i,0}, j >, 0)) = 0$
- $\triangleright\ f((< 2, \tau_{i,0}, 0 >, < 3, \tau_{i,0}, t >, 0)) = 0$
  $\triangleright f((< 2, \tau_{i,0}, j >, < 3, \tau_{i,0}, t >, 0)) = 0$
  $\triangleright$
  $\star\ f((< 2, \tau_i^j >, < 3, \tau_i, r_i^j >, 1)) = CH_i - (j - r_i^j.CH_i)$
  $\star\ f((< 2, \tau_i^j >, < 3, \tau_i, d_i^j - 1 >, 1)) = (J+1) - r_i^{j+1}.CH_i \text{ si } d_i^j - 1 = r_i^{j+1}$
  $\star$ *sinon* $f((< 2, \tau_i^j >, < 3, \tau_i, t >, 1)) = CH_i$
- $\triangleright\ f((< 3, \tau_{i,0}, t >, < 4, t >, 0)) = 0$
  $\triangleright f((< 3, \tau_i, t >, < 4, t >, 1)) = CH_i$
- $f((< 4, t >, < puit >, m)) = \displaystyle\sum_{\substack{\tau_i \in \tau t.q. \\ r_i + k.P_i \leq t < r_i + k.P_i + D_i}} CH_i$

Lemme 7. *f est un flot de valeur $\sum_{\tau_i \in \tau} DT_i(L)$ pour $G(L)$.*

Preuve du lemme

Nous prouvons tout d'abord que les contraintes de capacité sont bien respectées. Les arcs de E_0 et E_1 sont saturés et les arcs de E_3 transportent un flot de valeur égale soit à 0, soit à CH_i qui est inférieur ou égal à 1. Les contraintes de capacité sont donc respectées pour les arcs de E_0, E_1 et E_3. Si l'on considère un arc de E_4, il transporte un

flot de valeur $\sum_{\substack{\tau_i \in \tau \text{t.q.} \\ r_i + k.P_i \leq t < r_i + k.P_i + D_i}} CH_i \leq \sum_{\tau_i \in \tau} CH_i$, et comme nous avons, par hypothèse, $\sum_{\tau_i \in \tau} CH_i \leq m$, la contrainte de capacité est bien respectée. Enfin, pour les arcs de E_2, nous devons prouver que $CH_i - (j - r_i^j.CH_i) \leq 1$ et $(j+1) - r_i^{j+1}.CH_i \leq 1$ si $d_i^j - 1 = r_i^{j+1}$. Nous avons $r_i^j = \lfloor \frac{j}{CH_i} \rfloor$, donc $\frac{j}{CH_i} - 1 < r_i^j \leq \frac{j}{CH_i}$, donc $-CH_i < r_i^j * CH_i - j \leq 0$ et par suite $0 < CH_i - (j - r_i^j * CH_i) \leq CH_i \leq 1$. Nous pouvons prouver aussi que $(j+1) - r_i^{j+1} * CH_i \leq 1$ en utilisant le même raisonnement. Donc ici aussi, les contraintes de capacité sont bien respectées.

Nous prouvons ensuite que le flot f vérifie bien la loi des nœuds de Kirchhoff. Pour les nœuds ayant des entrées transportant un flot de valeur nulle, la loi est bien respectée car les arcs en sortie transportent aussi un flot nul. Pour chaque nœud $< 1, \tau_i >$ de V_1, le flot transporté par l'arc entrant a pour valeur $DT_i(L)$. Comme il y a $DT_i(L)$ arcs sortants saturés et que le reste des arcs sortants transportent un flot nul, alors la loi est bien vérifiée pour V_1. Chaque nœud $< 2, \tau_i^j >$ de V_2 a un arc entrant transportant un flot de valeur 1. Nous allons montrer que l'ensemble des arcs sortants transporte un flot de valeur 1 aussi. Le nombre d'arcs sortants de ce nœud est $d_i^j - r_i^j$. Alors le flot transporté par ces arcs a pour valeur :

si $d_i^j = r_i^{j+1}$, $CH_i - (j - r_i^j.CH_i) + CH_i.(d_i^j - r_i^j - 2) + (j+1) - r_i^{j+1}.CH_i$ qui vaut 1 après simplification ;

sinon nous avons $d_i^j - 1 \neq r_i^{j+1}$, donc $\lceil \frac{J+1}{CH_i} \rceil - 1 \neq \lfloor \frac{j+1}{CH_i} \rfloor$ ce qui signifie que $\frac{j+1}{CH_i}$ est un entier et par suite $d_i^j = \frac{j+1}{CH_i}$. Le flot transporté par les arcs a donc pour valeur $CH_i - (j - r_i^j.CH_i) + CH_i.(d_i^j - r_i^j - 1)$ qui vaut 1 après simplification. Là aussi la loi est bien respectée.

Au niveau de chaque nœud $< 3, \tau_i, t >$ de V_3, il y a un seul arc sortant qui transporte un flot de valeur CH_i. Nous allons donc montrer que le ou les arcs entrants dans ce nœud transporte(nt) un flot de valeur CH_i. En effet, si $d_i^j - 1 = r_i^{j+1}$, il y a deux arcs entrants dans le nœud qui transportent un flot de valeur $(j+1) - r_i^{j+1}.CH_i + CH_i - ((j+1) - r_i^{j+1}.CH_i) = CH_i$. Sinon, il y a un seul arc entrant dans le nœud et transportant un flot de valeur CH_i.

Considérons pour finir un nœud $< 4, t >$ de V_4. Les arcs entrant dans ce nœud et transportant un flot de valeur non nulle sont les arcs $(< 3, \tau_i, t >, < 4, t >, 1)$ avec $r_i + k.P_i \leq t < r_i + k.P_i + D_i$. Ces arcs transportent donc un flot de valeur $\sum_{\substack{\tau_i \in \Gamma \text{t.q.} \\ r_i + k.P_i \leq t < r_i + k.P_i + D_i}} CH_i$, qui est par définition égale est la valeur du flot transporté par l'unique arc sortant de ce nœud.

Nous avons donc prouvé que la loi des nœuds de Kirchhoff est bien vérifiée au niveau de chaque nœud. Par conséquent, f est un flot de valeur $\sum_{\tau_i \in \Gamma} DT_{\tau_i}(L)$. □

Nous pouvons terminer la preuve du théorème 5. Le lemme 7 implique l'existence d'un flot rationnel de valeur $\sum_{\tau_i \in \Gamma} DT_{\tau_i}(L)$ pour le graphe $G(L) = (V, E)$. Comme le flot tranporté par chaque arc a une valeur entière, alors il existe un flot de valeur entière $\sum_{\tau_i \in \Gamma} DT_{\tau_i}(L)$ dans le graphe $G(L)$ d'après [71]. Le lemme 7 prouve alors qu'un ordonnancement P-équitable peut être construit, d'où le théorème. □

5.2.3 Mise en œuvre

Afin de mettre en œuvre les algorithmes P-équitables dans notre contexte, nous ajoutons un état supplémentaire aux états d'une tâche : *l'état "oisif"*. A un instant t donné, une tâche τ_i est à l'état "oisif" si $t < r_i$ ou $t \in [k.T_i + r_i + D_i, (k+1).T_i)$. Dans ce cas la tâche n'a aucune sous tâche active. Quant à la politique d'attribution des priorités, à chaque instant, la priorité est donnée aux sous tâches de plus petite échéance. Les exaequo peuvent être départagés en utilisant *PF*, *PD* ou *PD*^{2} avec la définition suivante de *bit successeur* :

$$b_i^j = \begin{cases} 1 \text{ si } d_i^j > r_i^{j+1} \\ 0 \text{ sinon} \end{cases}$$

5.3 Pertinence de la condition et application

5.3.1 Pertinence de la condition

Pour illustrer la pertinence de nos résultats, nous avons procédé à des simulations. Notre objectif est d'étudier le pessimisme de notre condition. Pour cela, nous générons aléatoirement des échantillons de 1000 systèmes, dont nous contrôlons la charge CH ou le taux d'utilisation U. Nous simulons ensuite l'ordonnancement PF, afin d'obtenir le taux de systèmes ne vérifiant pas notre condition et étant cependant ordonnançables.

Le simulateur met en œuvre l'algorithme d'ordonnancement PF. Nous avons généré des tâches périodiques selon l'approche décrite à la page 63. La matrice utilisée pour générer les périodes afin de borner l'hyperpériode à 210 est :

$$\begin{pmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 3 \\ 1 & 1 & 5 & 5 \\ 1 & 1 & 7 & 7 \end{pmatrix}$$

L'intervalle d'étude pour un système de tâche est égal $[0,T)$ où T est égal à l'hyperpériode du système de tâche, pour des tâches à départs simultanés et $[0, max\{r_i\}_{i=1...n} + 2T)$ pour les systèmes de tâches à départs différés. L'intervalle d'étude pour les systèmes de tâches à départs différés est justifié par les résultats de nos travaux sur la cyclicité établis à la page 90.

Les résultats des simulations concernant l'ordonnançabilité sont résumés dans le tableau 5.I. Dans ce tableau, nous pouvons constater que conformément à nos résultats, tous les systèmes de tâches à échéances sur requêtes avec $U \leq m$ sont ordonnançables. De même, tous les systèmes de tâches à échéances contraintes avec $CH \leq m$ sont ordonnançables.

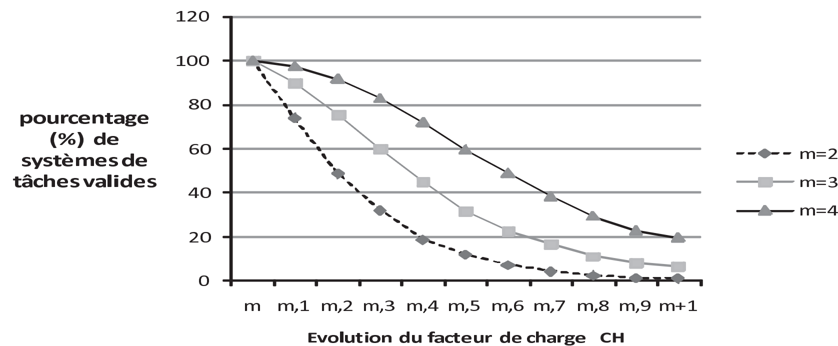
	Bornes de U et CH		
Systèmes de tâches	$U \leq m$	$CH \leq m$	$CH > m$
$r_i = 0$ et $D_i = T_i$	N=100%		
$r_i = 0$ et $D_i < T_i$	0< N < 100%	N=100%	0< N < 100%
$r_i > 0$ et $D_i = T_i$	N=100%		
$r_i > 0$ et $D_i < T_i$	0< N < 100%	N=100%	0< N < 100%

Tableau 5.I – Résultats de la simulation sur l'ordonnançabilité. N désigne le pourcentage de systèmes ordonnançable sur l'échantillon considéré.

Par ailleurs, nous pouvons aussi remarquer que :

1. Pour les systèmes de tâches à échéances contraintes, $U \leq m$ n'est pas une condition suffisante d'ordonnançabilité puisqu'il existe de tels systèmes de tâches non ordonnançables avec $U \leq m$;
2. Pour les systèmes de tâches à échéances contraintes, la condition $CH \leq m$ n'est pas une condition nécessaire il existe des systèmes de tâches avec $CH > m$ qui sont ordonnançables.

Nous avons donc affiné nos simulations pour voir l'impact de l'évolution du facteur d'utilisation U du système de tâches et du facteur de charge CH de ce système sur son ordonnançabilité. Sur la figure 5.3.1, nous avons représenté l'évolution du pourcentage de systèmes de tâches ordonnançables quand le facteur de charge évolue. Nous avons considéré des plateformes de $m = 2, m = 3$ ou $m = 4$ processeurs. Dans tous les cas, on observe que lorsque le facteur de charge vaut m, la totalité des systèmes de tâches générée est ordonnançable. Pour $CH \in]m, m+1[$ ($CH = m + \frac{k}{10}, 0 \leq k \leq 10$), le pourcentage de systèmes de tâches ordonnançables reste proche de 100% quand CH est proche de m et diminue rapidement quand CH tend vers $m+1$. Ces résultats montrent que la condition d'ordonnançabilité que nous avons établie est relativement bonne en ce sens qu'elle rejètera relativement peu de systèmes qui étaient en fait ordonnançables.



Pourcentage de tâches ordonnançables en fonction du facteur de charge sur une architecture de 2, 3 ou 4 processeurs.

Nous avons également examiné la corrélation entre l'évolution du facteur d'utilisation d'un système de tâches et son ordonnançabilité (dans le cadre des tâches à départs simultanés et à échéances contraintes). Ces résultats sont représentés sur les figures 5.3 et 5.4. Nous avons considéré une architecture de 3 processeurs pour la figure 5.3 et une architecture de 4 processeurs sur la figure 5.4. Nous pouvons observer que si U tend vers m, les systèmes de tâches générés ne sont pas ordonnançables et tous les systèmes de tâches deviennent ordonnançables quand $U \leq \frac{m}{2}$.

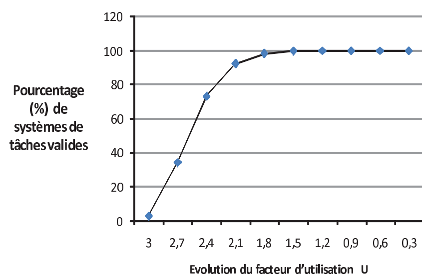


Figure 5.3 – Graphe 2 :

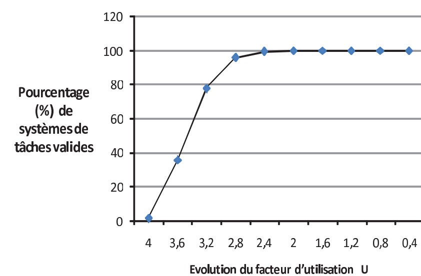


Figure 5.4 – Graphe 3 :

les graphes 2 et 3 montrent respectivement le pourcentage de tâches ordonnançables en fonction du facteur d'utilisation sur 3 processeurs et 4 processeurs.

5.3.2 Application

Les résultats obtenus dans ce chapitre vont être utilisés pour gérer les reprises après panne dans le cas de figure **S2-P1 et P2** de la page 105.
Soit $\tau = \{\tau_i(0, C_i, T_i, T_i), i = 1 \ldots n, n \in \mathbb{N}\}$ un système de tâches ayant un ordonnancement P-équitable sur m processeurs.
Si $\sum_{i=1}^{n} \frac{C_i}{T_i - C_i} \leq m$, alors il existe un ordonnancement P-équitable pour le système de tâches

$$\tau' = \left\{\tau_i'(0, C_i, T_i - C_i, T_i), i = 1 \ldots n, n \in \mathbb{N}\right\}.$$

De plus, $\forall \tau_i' \in \tau'$, τ_i' finit C_i unités de temps avant la fin de sa période courante.
Les figures 5.5 et 5.6 illustrent cela. Sur la figure 5.5, nous avons un ordonnancement *PF* sur 3 processeurs du système de tâches

$$\tau = \{\tau_1(0,3,6,6); \tau_2(0,2,6,6); \tau_3(0,5,12,12), \tau_4(0,4,12,12)\}.$$

La figure 5.6 est une séquence d'ordonnancement du système de tâches

$$\tau' = \{\tau_1(0,3,6-3,6); \tau_2(0,2,6-2,6); \tau_3(0,5,12-5,12); \tau_4(0,4,12-4,12)\} =$$

$$\{\tau_1(0,3,3,6), \tau_2(0,2,4,6), \tau_3(0,5,7,12), \tau_4(0,4,8,12)\}.$$

Chaque tâche de τ' finit au moins C_i unités de temps avant la fin de sa période. Sous l'hypothèse *H1* de la page 105, toute la tâche peut être reprise en cas de panne d'un processeur.

Conclusion du chapitre

Dans ce chapitre, nous avons étendu le principe de l'ordonnancement équitable à tout système de tâches périodiques. Nous avons donné une condition suffisante d'ordonnançabilité. Nous avons montré qu'il existe un ordonnancement P-équitable sur m processeurs pour tout système de tâches périodiques τ vérifiant $\sum_{i=1}^{n} \frac{C_i}{D_i} \leq m$. Nous avons

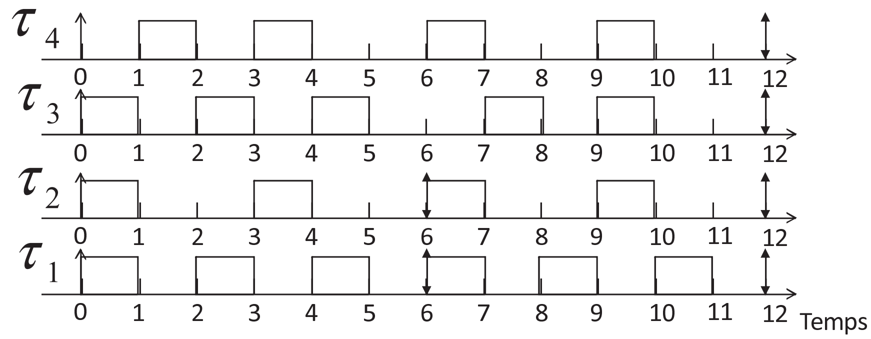


Figure 5.5 – Ordonnancement par PF du système τ sur 3 processeurs
$\tau = \{\tau_1(0,3,6,6), \tau_2(0,2,6,6), \tau_3(0,5,12,12), \tau_4(0,4,12,12)\}$

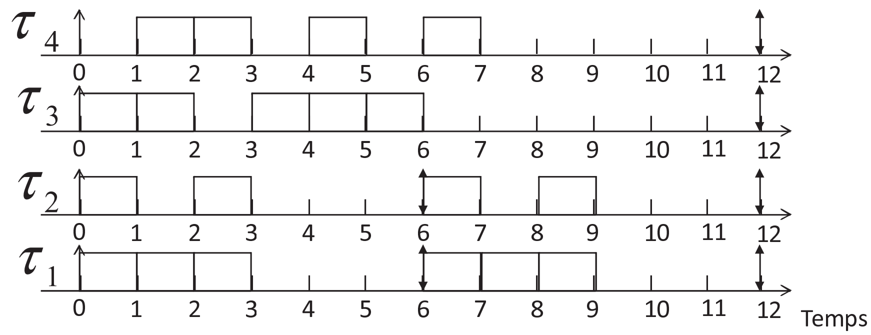


Figure 5.6 – Ordonnancement par PF du système τ' sur 3 processeurs
$\tau' = \{\tau_1(0,3,3,6), \tau_2(0,2,4,6), \tau_3(0,5,7,12), \tau_4(0,4,8,12)\}$

ensuite illustré la pertinence de nos résultats à travers des simulations. Ces simulations nous ont enfin permis de mesurer l'impact de la variation des facteurs d'utilisation et de charge du système sur l'ordonnaçablité du système par l'algorithme d'ordonnancement équitable PF.

CHAPITRE 6

GESTION DES TÂCHES APÉRIODIQUES

L'un des schémas de reprise après panne consiste en la reprise (éventuellement partielle) de la tâche fautive comme une tâche apériodique.
Un autre facteur d'évolution du système consiste en un changement de besoin. Cette évolution est généralement de nature fonctionnelle. Il s'agit d'étendre la capacité du système pour s'adapter à un nouvel environnement ou favoriser des interactions avec d'autres systèmes. Les fonctions existantes doivent alors être modifiées ou s'enrichir de nouvelles fonctions.
Nous étudions dans ce chapitre la problématique de l'ordonnancement des tâches apériodiques ainsi que l'intégration de nouvelles tâches dans le système. Cela peut constituer une réponse au problème de l'évolution aussi bien matérielle que fonctionnelle du système. Le travail que nous présentons dans ce chapitre s'appuie sur le travail de Master de Christian Fotsing, actuellement en thèse au LISI.

6.1 Le problème

6.1.1 Hypothèses sur les tâches

Nous considérons d'abord des tâches périodiques indépendantes à échéances sur requêtes, ordonnancées à l'aide d'une stratégie P-équitable. Nous considérons des tâches apériodiques à contraintes strictes, indépendantes entre elles et indépendantes des tâches périodiques. Une nouvelle tâche apériodique est traitée à son arrivée dans le système par une routine d'acceptation : elle est acceptée dans le système si et seulement si elle peut respecter son échéance et elle n'entraine pas le dépassement d'échéance, ni d'une tâche périodique, ni d'une tâche apériodique déjà acceptée. Les tâches apériodiques acceptées sont ordonnancées en arrière plan et nous proposons un test d'acceptation basé sur la répartition équitable des temps creux.
Nous étendons par la suite l'étude au cas où les tâches périodiques peuvent partager

des ressources ou être soumises à des relations de précédence et avoir des échéances contraintes. Dans ce cas, nous utilisons une stratégie d'ordonnancement à base de réseaux Petri. Nous ordonnançons hors-ligne les tâches périodiques en utilisant notre méthode à base de réseaux de Petri. La sélection d'une séquence se fera sur la base du critère de répartition équitable des temps creux.

6.1.2 Méthodes classiques d'ordonnancement des tâches apériodiques

Les techniques classiques d'intégration de tâches apériodiques dans le contexte des ordonnancements P-équitables consistent à considérer les tâches apériodiques comme de nouvelles tâches périodiques. Dans un contexte d'ordonnancement P-équitable sur une architecture de m processeurs et pour des tâches périodiques indépendantes à échéances sur requêtes, une nouvelle tâche apériodique τ_a de facteur d'utilisation U_a, est acceptée dans un système de facteur d'utilisation U, si $U_{total} = U + U_a \leq m$. Cette technique de prise en charge des tâches apériodiques est illustrée par la figure 6.1.

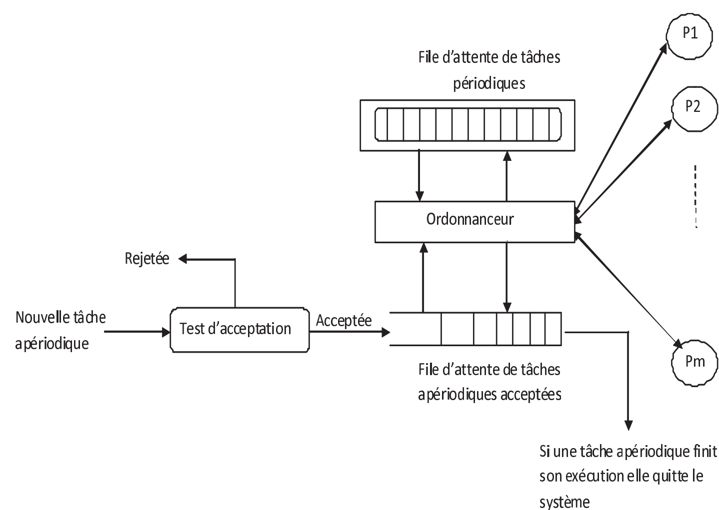


Figure 6.1 – Ordonnancement des tâches apériodiques : méthode conjointe
Les tâches apériodiques sont placées dans la même file que les tâches périodiques et sont traitées de la même façon par l'ordonnanceur

Dans la littérature, la mise en œuvre de ces techniques se fait de deux façon [108] : la première approche consiste à estimer le temps de réponse du flux d'apériodiques qui arrivent dans le système, ce qui permet de savoir si l'ordonnançabilité est garantie, et dans la seconde approche, on modélise les tâches périodiques par des tâches *GIS* (une sous tâche peut s'exécuter plus tôt que prévu ou encore son exécution peut être retardée). Pour s'assurer de l'ordonnançabilité dans la seconde approche, il suffit alors de s'assurer que l'ensemble des tâches (tâches apériodiques et périodiques) vérifient la condition d'ordonnançabilité $U_{total} \leq m$. L'avantage d'une telle approche est que le test d'acceptation d'une nouvelle tâche apériodique est de complexité en $O(1)$. Mais le modèle de tâche considéré n'est pas adapté à notre étude (dans notre approche des ordonnancements P-équitables, nous considérons le modèle de tâche classique de Liu et Layland et toutes les sous tâches s'exécutent dans leur fenêtre d'exécution) d'où son abandon.
La première approche est adaptée à notre étude mais sa mise en œuvre est complexe du fait de la complexité de son test d'acceptation. En effet, la complexité du test d'acceptation de cette approche est $O(klogk + (n_a + k)f)$ où k est le nombre de tâches apériodiques qui arrivent dans le système, n_a est le nombre de tâches apériodiques déjà acceptées en plus de celles qui viennent d'arriver, et en enfin f est la complexité de la fonction de calcul du temps de réponse des tâches apériodiques. La complexité élevée de cette approche est due au fait que les nouvelles tâches apériodiques qui arrivent sont a priori acceptées et on calcule le temps de réponse du nouvel ensemble de tâches apériodiques. Si l'ensemble des tâches apériodiques nouvellement construit n'est pas ordonnançable, alors on élimine au fur et à mesure certaines tâches apériodiques. On ne peut cependant pas éliminer une tâche apériodique qui avait déjà été acceptée.
Dans le cadre des ordonnancements hors-ligne à base de modèle, très peu de travaux se sont intéressés à l'ordonnancement des tâches apériodiques du fait de la rigidité de ce modèle d'ordonnancement.
Notre objectif est de proposer un critère plus fin d'acception des tâches apériodiques avec une complexité raisonnable dans le cadre des ordonnancements P-équitables. Dans le cadre des ordonnancements hors-ligne à base de réseaux de Petri, nous allons proposer un critère de sélection de séquences d'ordonnancement avec une répartition équitable

des temps creux. Cela permettra d'utiliser le critère d'acception proposé dans le cadre des ordonnancements P-équitables. Ce critère d'acceptation exploite la prédictibilité des ordonnancements P-équitables. Le nombre de temps creux dont disposent les tâches apériodiques pour leur exécution est connu à deux unités de temps près, ce qui permet de rejeter sans calcul supplémentaire une nouvelle tâche apériodique.
Le test que nous proposons est un test de complexité polynomiale et nous illustrerons à travers des simulations que ce test présente bien de meilleurs résultats.

6.2 Principe général - Gestion des temps creux

Nous nous limitons pour le moment au cas où $m-1<U<m$. L'ordonnancement des tâches apériodiques en arrière plan a comme atout majeur de ne pas remettre en cause l'ordonnancement périodique. On n'a donc pas besoin de recalculer l'ordonnancement périodique à cause d'une tâche apériodique prise en compte dans le système et par voie de conséquence, l'ordonnancement périodique peut être fait hors-ligne.
L'ordonnancement en arrière plan des tâches apériodiques repose sur l'évaluation du nombre de temps creux dans un intervalle de temps quelconque en temps constant avec une erreur limitée. Il est nécessaire pour cela que les temps creux soient équitablement répartis.
Les temps creux sont les temps d'inactivité laissés par l'ordonnancement des tâches périodiques. La solution la plus simple consiste à ordonnancer les tâches périodiques selon la stratégie choisie, les temps creux se plaçant chaque fois que moins de m tâches sont ordonnancées. Mais ceci a comme inconvénient le fait qu'on n'a aucune maîtrise sur la répartition des temps creux. Une solution alternative consiste à utiliser le principe de complétion vu au chapitre 4, mais ici, nous introduisons une unique tâche oisive τ_0. Comme nous avons supposé que $m-1 \leq U$, nous avons $C_0 \leq T$ et ici nous voulons interdire le parallélisme. La tâche τ_0 gère l'ensemble des temps creux, et si nous contrôlons son exécution, nous contrôlons la répartition des temps creux. La tâche oisive a pour

paramètres

$$\tau_0 =< 0, T(m-U), T, T >$$

et son ordonnancement à instant t correspond à un temps creux. La figure 6.2 illustre l'ordonnancement par l'algorithme PF du système de tâche

$$\tau =< \tau_1(0,3,6,6), \tau_2(0,2,6,6), \tau_3(0,5,12,12), \tau_4(0,5,12,12) >$$

sans la tâche oisive. Nous constatons que la répartition en arrière plan des temps creux n'est pas équitable. Il y a par exemple deux temps creux simultanés à l'instant $t = 11$, et aucun temps creux n'a eu lieu au bout de 3 unités de temps alors qu'une répartition équitable l'exigerait. Sur la figure 6.3, le même système de tâche a été complété par la tâche oisive $\tau_0(0,4,12,12)$ et la politique d'ordonnancement PF est utilisée. La tâche τ_0 est ordonnancée selon PF, donc les temps creux sont répartis équitablement.

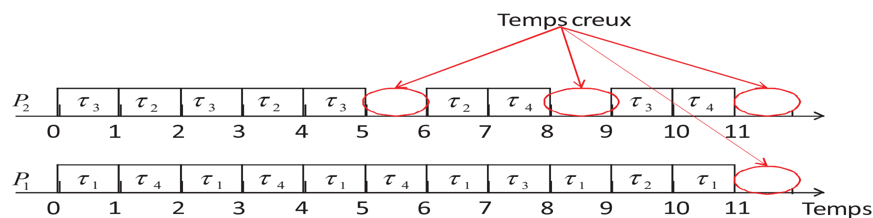


Figure 6.2 – Ordonnancement du système de tâche τ sans la tâche oisive

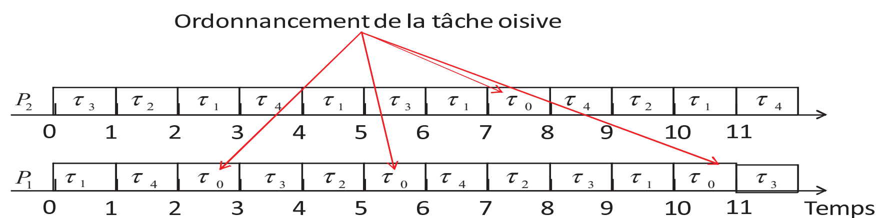


Figure 6.3 – Ordonnancement du système de tâche τ avec la tâche oisive

La tâche oisive sert de serveur de tâches apériodiques dans notre étude. Dès que la

tâche oisive est ordonnancée, une tâche apériodique peut être exécutée. Si aucune tâche apériodique n'est présente, alors un processeur demeure effectivement inactif. Nous supposons que la file des tâches apériodiques est triée par EDF [81]. Dans ce cas, la tâche d'échéance la plus proche s'exécute quand une tâche apériodique est ordonnancée. En cas d'égalité, la priorité est donnée à la tâche la plus ancienne.
La prédictibilité des ordonnancements P-équitables permet d'avoir un test d'acception de tâches apériodiques de complexité minimale. En effet, cette prédictibilité permet d'avoir entre deux instant t et t' une approximation de la capacité du serveur (le nombre de temps creux disponibles). Considérons l'intervalle de temps $[t,t']$. L'ordonnancement étant P-équitable, nous avons

$$\begin{cases} \lfloor U_0 t \rfloor \leq W_0(0,t) \leq \lceil U_0 t \rceil \\ \lfloor U_0 t \rfloor \leq W_0(0,t) \leq \lceil U_0 t \rceil \end{cases}$$

Nous en déduisons alors,

$$\lfloor U_0 t' \rfloor - \lceil U_0 t \rceil \leq W_0(t,t') \leq \lceil U_0 t' \rceil - \lfloor U_0 t \rfloor .$$

Avec cette relation, nous avons une valeur minimale pour le nombre d'unités de temps dont dispose une tâche apériodique pour son ordonnancement. En effet, nous pouvons approximer ce nombre par $\lfloor U_0 t' \rfloor - \lceil U_0 t \rceil$ avec une erreur d'approximation de deux unités de temps au maximum. Nous noterons $W_M(t,t')$ cette valeur minimale dans la suite de notre étude.

6.3 Le protocole d'acceptation

Nous rappelons que les tâches apériodiques prises en charge (déjà acceptées) sont départagées selon une politique d'ordonnancement EDF. Soit $(\tau_{a_1}, \tau_{a_2}, \ldots, \tau_{a_k})$ l'ensemble des tâches apériodiques déjà acceptées et non terminées. Nous supposons ces tâches ordonnées par ordre croissant d'échéances c'est à dire $d_{a_1} \leq d_{a_2} \leq, \ldots, \leq d_{a_k}$. Soit $\tau_s =< t, C_a, D_a >$ une nouvelle tâche apériodique qui arrive à l'instant t. Son échéance

est alors $d_a = t + D_a$. L'acceptation de la tâche dépend du temps processeur libre entre l'instant t et de son échéance d_a. Le test d'acceptation est ainsi défini :

- si $d_{a_k} \leq d_a$, alors la nouvelle tâche apériodique τ_a a la plus grande échéance. Alors son adjonction à la liste n'aura aucun impact sur l'exécution des autres tâches apériodiques existantes. La décision d'acceptation dépend seulement du nombre d'unités de temps processeur libres entre t et d_a. S'il y a assez de temps libre pour que τ_a puisse s'exécuter après les autres tâches apériodiques de la liste, alors elle est acceptée ;sinon elle est rejetée. Formellement, τ_a est acceptée si et seulement si : $W_M(t,d_a) \geq \sum_{i=1}^{i=k} RCT_{a_i}(t) + C_a$;
- si $\exists j \in 1 \ldots k-1$ tel que $d_{a_j} \leq d_a < d_{a_{j+1}}$, les j premières tâches apériodiques ne seront pas affectées par l'acceptation de τ_a. Par contre, l'acception de τ_a va décaler l'exécution de $\tau_{a_{j+1}}, \ldots, \tau_{a_k}$. Nous devons donc nous assurer que l'acception de τ_a ne provoque pas de dépassement d'échéance pour l'une des tâches apériodiques $\tau_{a_{j+1}}, \ldots, \tau_{a_k}$. Formellement, τ_a est acceptée si et seulement si :
  1. si τ_a a assez de temps processeur libre pour s'exécuter :
     $W_M(t,d_a) \geq \sum_{i=1}^{i=j} RCT_{a_i}(t) + C_a$;
  2. chaque tâche décalée aura assez de temps processeur libre pour s'exécuter :
     $\forall p \in j+1 \ldots k,\ W_M(t,d_{a_p}) \geq \sum_{i=1}^{i=p} RCT_{a_i}(t) + C_a$
- si $d_a < d_{a_1}$, alors l'acceptation de τ_a va décaler l'exécution de toutes les autres tâches apériodiques de la liste. Alors, nous devons vérifier que :
  1. τ_a a assez de temps processeur libre pour s'exécuter :$W_M(t,d_a) \geq C_a$;
  2. les autres tâches apériodiques ont également assez de temps processeur libre pour leur exécution : $\forall p \in 1 \ldots k, W_M(t,d_{a_p}) \geq \sum_{i=1}^{i=p} RCT_{a_i}(t) + C_a$.

Pour la mise en œuvre pratique du serveur, une table de tâches apériodiques est maintenue triée par ordre croissant d'échéance (voir le tableau 6.I). Pour chaque tâche apériodique on stocke son échéance d_a, le temps processeur qu'il lui faut pour terminer son exécution RCT_a ainsi que le temps processeur cumulé qu'il lui faut à elle et à toutes les tâche apériodiques qui la précèdent C_RCT_a.

	Paramètres dynamiques des tâches apériodiques			
Identifiant	τ_{a_1}	τ_{a_2}	...	τ_{a_k}
Échéance	d_{a_1}	d_{a_2}	...	d_a
RCT	RCT_{a_1}	RCT_{a_2}	...	RCT_{a_k}
C_RCT	RCT_{a_1}	$RCT_{a_2}+RCT_{a_1}$	...	$RCT_{a_1}+$ $\ldots+RCT_{a_k}$

Tableau 6.I – Table des paramètres dynamiques des tâches apériodiques

6.4 Analyse de performance

Nous évaluons tout d'abord la complexité de notre approche. Comme l'ordonnancement périodique n'est pas remis en cause lors de l'acceptation d'une tâche apériodique, il peut être calculé une fois pour toutes hors ligne avant l'exécution sur une hyperpériode. La complexité de notre approche dépend donc uniquement de la complexité du calcul pour déterminer le temps processeur libre et de la complexité du test d'acceptation. La première étape a une complexité en $O(1)$ et la deuxième étape nécessite $O(k)$ additions et $O(k)$ comparaison où k est le nombre de tâches apériodiques en cours. Enfin la mise à jour de la liste est de complexité $O(k)$.
Nous avons ensuite mené des simulations pour comparer notre approche à l'approche proposée dans [108]. Nous utiliserons une *approche exacte* qui sert de référence pour la comparaison. Dans l'approche exacte, l'acceptation d'une tâche apériodique dépend du nombre exact d'unités de temps processeurs disponible. L'élément de comparaison est le taux de charge apériodique acceptée. Avant de présenter les résultats, nous présentons les conditions de simulation. Plusieurs facteurs peuvent influencer l'ordonnancement des tâches apériodiques. Il s'agit du nombre de processeurs de la plateforme, du facteur d'utilisation total des processeurs par les tâches périodiques et de la loi d'arrivée des tâches apériodiques. Nous faisons varier ces paramètres pour observer le taux de tâches apériodiques acceptées.
Les systèmes de tâches périodiques générés sont composés de tâches à départs simultanés et à échéances sur requête, ce qui fait que la longueur de la séquence à observer est $l'hyperpériode\ T$ pour chaque système de tâches. Pour nos illustrations nous générons

des échantillons de 500 systèmes de tâches périodiques. Chaque échantillon $S(U,m,i)$, $i=1,\ldots,9$ (nous générons une dizaine d'échantillons) est caractérisé par son facteur moyen d'utilisation qui est compris dans l'intervalle $\left[m-1+\frac{i}{10},m-1+\frac{i+1}{10}\right]$ où m est le nombre de processeurs de la plateforme.
Pour chaque système de tâches périodiques, nous générons un flot de tâches apériodiques. La loi d'arrivée des apériodiques est modélisée par une loi exponentielle de paramètre x. Le délai D_{a_i} d'une tâche apériodique τ_{a_i} est une variable aléatoire uniforme comprise dans l'intervalle $[10,D_Max]$ et sa durée d'exécution est une variable aléatoire uniforme dans l'intervalle $\left[\frac{D_{a_i}}{10},\frac{D_{a_i}}{2}\right]$.
Pour chaque échantillon, nous avons effectué trois simulations sur l'intervalle d'étude $[0,T)$:

1. nous utilisons notre méthode, les tâches périodique sont ordonnancées selon l'algorithme d'ordonnancement équitable PF ;
2. nous utilisons la méthode proposée par [108] (la méthode conjointe) qui consiste à accepter une nouvelle tâche apériodique si le système n'est pas surchargé i.e $U_{total}=U+U_a\leq m$;
3. nous utilisons la méthode exacte basée sur un comptage exact du nombre de temps creux disponibles dans l'ordonnancement périodique. Nous procédons donc comme dans notre méthode sauf qu'ici le nombre de temps creux disponible n'est plus approximé.

Après les simulation nous procédons à une analyse compétitive avec comme méthode de référence, la méthode exacte. Pour chaque couple

$$(\text{système de tâches périodiques, flux de tâches apériodique}),$$

nous comparons les ratios

$$\frac{\text{charge apériodique acceptée par notre méthode}}{\text{charge apériodique acceptée par la méthode exacte}} \text{ et}$$

$$\frac{\text{charge apériodique acceptée par la méthode conjointe}}{\text{charge apériodique acceptée par la méthode exacte}}.$$

Nous avons effectué des simulations pour plusieurs valeurs du triplet (m,x,D_Max). La figure 6.4 montre les résultats pour $(4,40,200)$. Les résultats pour $(2,20,40)$, $(4,40,40)$ conduisent aux mêmes conclusions.

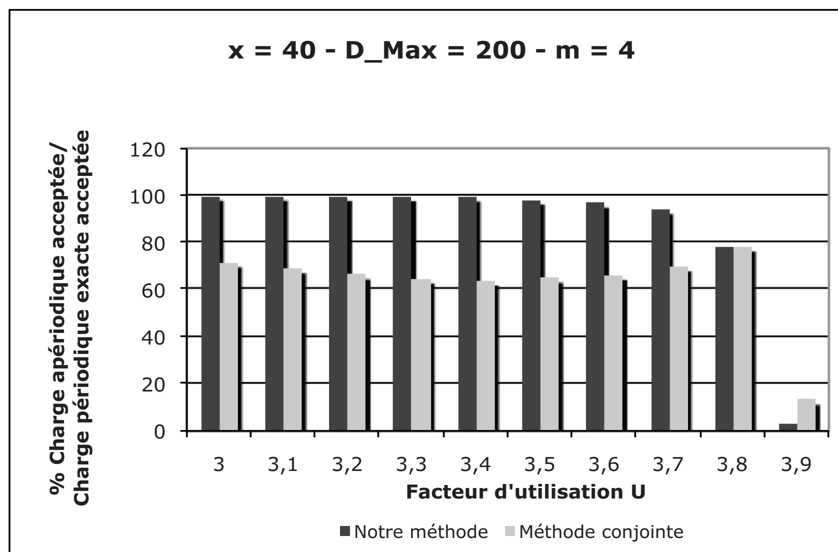


Figure 6.4 – Comparaison de notre méthode à la méthode conjointe

Nous pouvons constater que pour pratiquement toutes les valeurs de U, notre approche a des résultats très proches de ceux obtenus par l'approche exacte, et ils sont meilleurs que ceux obtenus par la méthode conjointe. La performance de notre approche diminue sensible pour les valeurs de U telles que $m-0.1 < U < m$. Cela s'explique par le fait que pour de telles valeurs de U, le temps processeur disponible pour l'exécution de tâches apériodiques diminue énormément ce qui fait que l'approximation que nous faisons devient sensible. Nous obtenons les mêmes résultats en monoprocesseur ce qui nous fait dire que notre approche tend vers l'optimalité.

Nous étendons maintenant notre test d'acception à un contexte plus large où les tâches peuvent être à départs différés, à échéances contraintes, partager des ressources

et avoir entre elles des relations de précédence. La principale exigence de notre test est la répartition équitable des temps creux. Nous allons donc adapter l'approche à base de modèle que nous proposons afin de sélectionner les séquences pour lesquelles les temps creux sont équitablement répartis.

6.5 Généralisation

Dans cette section nous nous intéressons aux tâches pouvant partager de ressources critiques ou communiquer entre elles. Les tâches peuvent également être à départs différés et à échéances contraintes. Dans ce contexte, il n'est plus possible d'utiliser les techniques P-équitables pour ordonnancer les tâches périodiques. Par ailleurs, nous n'avons pas besoin que toutes les tâches périodiques soient ordonnancées de façon équitable. Seule la tâche oisive doit l'être. Nous allons donc utiliser les techniques d'ordonnancement hors-ligne afin de sélectionner des séquences pour lesquelles les temps creux sont répartis équitablement (s'il existe de telles séquences), mais par contre, nous n'imposerons rien aux tâches périodiques. L'un des atouts des approches hors-ligne est de permettre précisément cette souplesse.
Rappelons que la performance de notre test d'acceptation repose sur la répartition équitable des temps creux. Nous utilisons un ordonnancement hors-ligne à base de réseau de Petri, qui est adapté au modèle de tâches que nous considérons.
Quand cette technique d'ordonnancement est utilisée, l'ensemble des ordonnancement valides est l'ensemble des chemins du graphe des marquages accessibles qui est un graphe orienté. La recherche d'une séquence optimale est équivalente à la recherche d'un chemin optimal dans un graphe. Cela se fait en pondérant les arcs du graphe et en recherchant un chemin de coût minimum ou maximum dans le graphe ainsi valué. Le poids attribué aux arcs dépend du critère d'optimisation choisi.

6.5.1 Compléments sur l'approche à base de réseaux de Petri

Rappelons que l'ensemble des séquences d'ordonnancement valides est donné par le graphe des marquages accessibles. Un chemin du graphe des marquages est une sé-

quence d'ordonnancement valide du système modélisé. Chaque transition allant d'un marquage à un autre est étiquetée par un multi-ensemble ω_h de cardinalité inférieure ou égale m représentant les tâches choisies. Si la profondeur du graphe est L, chaque chemin sera un mot de la forme

$$\sigma = \omega_1 \omega_2 \ldots \omega_L \text{ où } \omega_h = \left\{ \tau_{i_1} \ldots \tau_{i_{m'}} \right\}, \ avec \ m' \leq m$$

ce qui signifie que les tâches $\left\{ \tau_{i_1} \ldots \tau_{i_{m'}} \right\}$ se voient attribuer les processeurs à l'instant $t = h$.

6.5.2 Extraction des séquences avec temps creux équitablement répartis

Nous avons vu dans les sections précédentes que lorsque le système de tâches n'est pas de charge maximale, des temps creux cycliques peuvent apparaître dans l'ordonnancement. Même pour les systèmes de charge maximale, des temps creux acycliques apparaissent durant la montée en charge du système. Pour modéliser les temps creux, [60] regroupe les temps creux acyclique au sein d'une tâche appelée *tâche creuse* notée τ_c et les temps creux cycliques sont regroupés au sein de la *tâche oisive*, τ_0. La *tâche creuse* est alors utilisée juste pour représenter les temps creux durant la montée en charge du système et est active entre 0 et $t_c - 1$. La tâche oisive est quand à elle activée à t_c. Cela est possible parce que le temps de montée en charge est connu dans le cas monoprocesseur [37]. Dans le cas multiprocesseur le temps de montée en charge n'est pas connu et peut être très tardif [38]. Cela nous a conduit à ne considérer que les temps creux cycliques régroupés au sein de la tâche oisive.

6.5.2.1 Modélisation de la tâche oisive

Nous souhaitons que la tâche oisive soit ordonnancée de façon équitable. Une première solution consiste à la décomposer en C_0 sous tâches

$$\left\{\tau_{0_i} = (\left\lfloor\frac{i}{U_0}\right\rfloor, 1, \left\lceil\frac{i+1}{U_0}\right\rceil - \left\lfloor\frac{i}{U_0}\right\rfloor, T) i = 1, \ldots, C_0\right\}$$

Dans ce cas, dans chaque séquence valide trouvée, chaque sous tâche est ordonnancée dans sa fenêtre d'exécution, donc la tâche oisive est ordonnancée de manière P-équitable. Cette technique donne des solutions exactes, mais elle a pour principal défaut d'augmenter fortement et la taille du réseau de Petri (car on ajoute un grand nombre de tâches) et les temps de calcul (car l'adjonction de tâches induit de très nombreux retours en arrière supplémentaires qui sont très coûteux). L'alternative consiste à garder une seule tâche oisive, et à utiliser une valuation du graphe des marquages qui permettra d'extraire les séquences les meilleures au sens de l'équité.

Contrairement au cas monoprocesseur, la tâche oisive peut se paralléliser dans le contexte multiprocesseur. Elle sera donc représentée par m transitions parallèles mises en exclusions mutuelles. A chaque hyperpériode T, $T(m-U)$ jetons sont produits dans $Activ_0$, correspondant à l'activation de la tâche oisive. Chaque jeton représente un temps creux qui a lieu dans l'hyperpériode. Le tir de la transition $T_{0,1,i}$ pour i allant de 1 à m, signifie que i processeurs sont oisifs car i jetons processeur sont utilisés. La transition $T_{0,1,i}, i = 1 \ldots m$ sera donc étiquetée par le multi-ensemble $\{\tau_0\}^i$. La modélisation de la tâche oisive est représentée sur la figure 6.5. La figure 6.6 illustre une modélisation de la tâche oisive sans parallélisation.

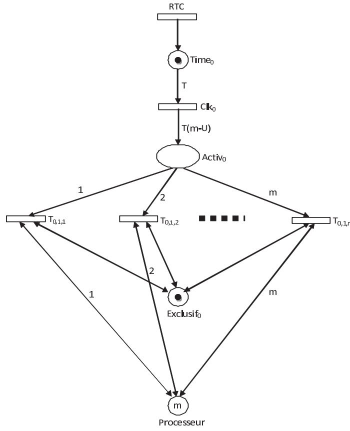


Figure 6.5 – Tâche oisive avec parallélisation

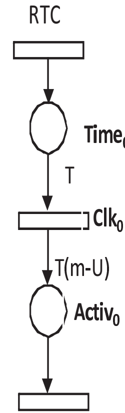


Figure 6.6 – Tâche oisive sans parallélisation

6.5.2.2 Ajout du critère de selection des séquences avec temps creux équitablement répartis

La répartition équitable idéale des temps creux implique qu'à chaque instant t, on ait

$$\overline{RCT_0}(t) = \begin{cases} k*C_0 + \frac{C_0}{D_0}*(t-k*T_0)\ si \\ t \in [k*T_0, k*T_0+D_0) \\ \\ (k+1)*C_0 \\ si\ t \in [k*T_0+D_0, (k+1)*T_0). \end{cases}$$

$k = \left\lfloor \frac{t}{T_0} \right\rfloor$ représente le numéro de l'instance en cours. Une illustration est donnée sur la figure 6.7.

Afin de sélectionner sur le graphe un ordonnancement sur lequel les temps creux seraient équitablement répartis, la valuation du graphe est telle que chaque arc est pondéré par la partie entière de l'écart absolu entre la répartition idéale et la répartition réelle. Soit (N_i, N_j) un arc à la profondeur h tel que $\tau_0 \in \omega_h$. Nous notons M_h le marquage obtenu après le tir groupé du groupe de transitions contenant τ_0. Alors le poids de l'arc est

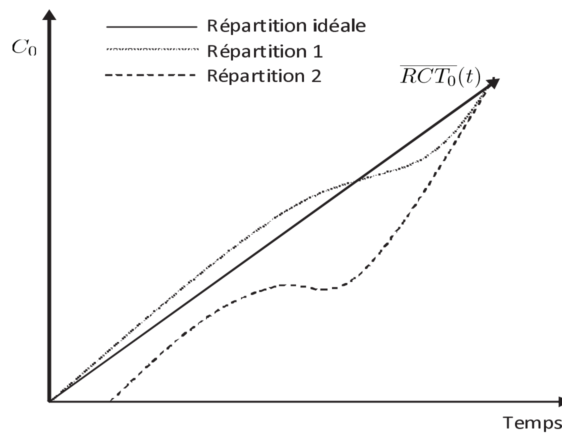


Figure 6.7 – Répartition des temps creux :
cette figure illustre deux répartitions possibles des temps creux autour d'une répartition idéale. Le choix que nous faisons dans notre étude c'est la répartition qui se rapproche de l'idéal. Cela correspond à la répartition 1 sur la figure.

défini par :

$$coût_arcTC(N_i,N_j) = \begin{cases} \lfloor |T(m-U) - M_h(Activ_0) - (h\ mod\ T)(u_0)| \rfloor \text{ si } \tau_0 \in \omega_h \\ 0 \text{ sinon} \end{cases}$$

Dans cette expression, $T(m-U) - M_h(Activ_0)$ correspond au nombre de temps creux déjà utilisés dans l'hyperpériode courante et $(h\ mod\ T)(u_0)$ est le nombre de temps creux qui devraient être idéalement utilisés dans l'hyperpériode.
L'évaluation du coût d'un nœud N se calcule alors de la façon suivante :

$$Coût_neoudTC(N) = min_{N' \in fils(N)} \left\{ Coût_neoudTC(N') + coût_arcTC(N,N') \right\}$$

Ce coût minimise l'écart entre la répartition équitable idéale et la répartition réelle des temps creux. Si nous avons un chemin de poids nul, alors cela correspond à une séquence où les temps creux sont répartis de façon équitable.

6.5.3 Autres critères

Rappelons que le problème abordé dans notre étude est la reprise d'une tâche en cas de panne d'un processeur. Pour que l'instance d'une tâche puisse être reprise et terminer avant son échéance, cette instance doit terminer le plus tôt possible, d'où la nécessité de minimiser le temps de réponse des tâches. Nous considérons donc la *minimisation du temps de réponse* des tâches comme un critère de performance à utiliser.
De plus, pour une prévisibilité plus fine du temps de réponse des tâches, il faut minimiser leur gigue d'activation. Nous sélectionnons alors les séquences minimisant le temps de réponse de tâches périodiques ainsi que leur gigue d'activation. La *minimisation de la gigue* des tâches est donc un critère de performance que nous prenons en compte.

6.5.3.1 La minimisation du temps de réponse

Ce critère s'applique sur l'ensemble des tâches de la configuration sauf la tâche oisive. Chaque arc est étiqueté par un multi-ensemble correspondant à l'ensemble des tâches ayant bénéficié des processeurs à cet instant. Le poids d'un arc sera la somme des contributions de chaque tâche du multi-ensemble à l'arc.
Considérons deux marquages N_i et N_j reliés par l'arc $Arc(N_i, N_j)$ à la hauteur h, et étiqueté par le multi-ensemble w_h. La contribution $p_k, k = 1 \dots m$ de chaque tâche τ_k du multi-ensemble à l'arc correspond au temps de réponse de l'occurrence de τ_k à la hauteur h. On repère donc les arcs qui mènent à la terminaison de certaines instances, et pour chaque instance, on fait apparaître son temps de réponse. Numériquement p_k est défini par :

$$p_k = \begin{cases} 0 \text{ si l'arc ne correspond pas à une terminaison de } \tau_k \\ \text{sinon} \begin{cases} T_k & \text{si } h - r_k \equiv 0\,[T_k] \\ h - \left\lfloor \frac{h-r_k}{T_k} \right\rfloor T_k - r_k & \text{sinon} \end{cases} \end{cases}$$

L'arc est donc pondéré par $co\hat{u}t_arcTR(N_i, N_j) = \sum_{\tau_k \in w_h} p_k$. Le coût d'un noeud N

est alors donné par

$$Coût_noeudTR(N) = min_{N_i \in fils(N)} \{Coût_noeudTR(N_i) + coût_arcTR(N,N_i)\}.$$

Un chemin de poids minimal est un chemin qui minimise le temps de réponse moyen des tâches.

6.5.3.2 La minimisation de la gigue temporelle

Le phénomène de *gigue temporelle* est le terme employé pour définir l'*irrégularité d'exécution* d'une tâche périodique. Cette irrégularité peut être évaluée de plusieurs façons en terme de *gigue de début d'exécution* et de *gigue de fin d'exécution*. Notons $s_{i,k}$ (respectivement $e_{i,k}$) les dates de début d'exécution (respectivement de fin d'exécution) de la $k^{ième}$ instance de la tâche $\tau_i \in \tau$. Les écarts $\delta s_{i,k}$ (respectivement $\delta e_{i,k}$) entre deux débuts d'exécution (respectivement en deux fins d'exécution) concernant deux instances successives d'une tâche τ_i sont définis par : $\delta s_{i,k} = s_{i,k+1} - s_{i,k}$ (respectivement $\delta e_{i,k} = e_{i,k+1} - e_{i,k}$). La minimisation de la gigue veut rendre ces écarts constants égaux à T_i. Dans un contexte d'ordonnancement en ligne, cela se fait en éliminant les causes de ces écarts (modification des dates de première activation des tâches et de leur priorité [45]).

Dans notre étude nous nous intéressons à la minimisation de la gigue d'activation. Notre approche consiste à étudier le graphe des marquages accessibles afin d'en extraire les séquences de gigue d'activation minimale pour l'ensemble des tâches hormis la tâche oisive.

Soit τ_i une tâche dont on veut minimiser la gigue d'activation. La tâche τ_i a $\left\lfloor \frac{L-r_i}{T_i} \right\rfloor$ instances et la date d'activation de la $k^{ième}$ est donnée par $r_{i,k} = r_i + (k-1)T_i$. Minimiser la gigue d'activation de τ_i revient à minimiser l'écart des différentes activations au plus tôt des instances de τ_i.

Pour avoir les exécutions au plus tôt de la tâche τ_i, le coût d'un arc (N_i, N_j) à la profon-

deur h est donné par :

$$coût_arcAuplustot_i(N_i,N_j) = \begin{cases} h \text{ si } \tau_i \in \omega_h \wedge (\text{t=h correspond à un instant de début} \\ \text{d'exécution d'une nouvelle instance } de\ \tau_i) \\ 0 \text{ sinon} \end{cases}$$

Pour la tâche τ_i, plus sa contribution au poids de l'arc est élevée, plus l'exécution de l'instance concernée a été tardive.
Posons $k = \left\lfloor \frac{h-r_i}{T_i} \right\rfloor$. Pour avoir la séquence minimisant la gigue d'activation de τ_i, l'arc (N_i,N_j) sera pondéré par :

$$coût_arcGigue_i(N_i,N_j) = \left| coût_arcAuplustot_i(N_i,N_j) - r_{i,k} \right|.$$

Cette valeur mesure l'écart entre la date de début d'exécution réelle de l'instance de la tâche τ_i et la date d'activation prévue de cette même instance.
L'arc (N_i,N_j) étant étiqueté par plusieurs tâche, le poids de l'arc doit contenir la contribution de chaque tâche l'étiquetant. Posons $\omega_h = \left\{ \tau_{i_1}, \ldots, \tau_{i'_m} \right\}$ avec $m' \leq m$. La contribution de la tâche $\tau_{i_j} \neq \tau_0$ au poids de l'arc est

$$coût_arcGigue_{i_j}(N_i,N_j).$$

Le coût de l'arc (N_i,N_j) est donc

$$coût_arcGigue(N_i,N_j) = \sum_{\tau_{i_j} \neq \tau_0 \in \omega_h} coût_arcGigue_{i_j}(N_i,N_j).$$

L'évaluation du coût d'un nœud N se calcule alors de la façon suivante :

$$Coût_noeudGigue(N) = min_{N' \in fils(N)} \left\{ Coût_noeudGigue(N') + coût_arcGigue(N,N') \right\}$$

Ce coût minimise les écarts entre les dates d'activations réelles et prévues des instances des tâches. Un chemin de coût nul donne une séquence où toutes les dates d'activation

sont bien respectées.

Conclusion du chapitre

Dans ce chapitre nous avons proposé une approche de complexité en $O(k)$, où k est le nombre de tâches apériodiques présentes dans le système, pour la prise en charge de tâches apériodiques dans un contexte d'ordonnancement P-équitable. Le test que nous avons proposé tire profit de la répartition équitable des temps d'inactivité des processeurs.
La même approche a été utilisée pour l'étude de la prise en charge des tâches apériodiques dans un contexte plus général où les tâches périodiques initiales peuvent avoir des relations de précédence ou partager des ressources critiques. Pour ce modèle de tâches, nous avons utilisé une technique d'ordonnancement hors-ligne à base de réseaux de Petri car c'est l'approche la mieux adaptée pour ce modèle de tâches.

CHAPITRE 7

CONCLUSION ET PERSPECTIVES

7.1 Conclusion

Dans cette thèse, nous avons mis en place des résultats nécessaires à l'étude de l'évolutivité des applications temps-réel en environnement multiprocesseur. Nous avons premièrement considéré le problème de la panne processeur qui est un évènement très probable dans ce contexte. Comme solution, nous avons proposé une technique de redondance temporelle qui consiste en la reprise de l'instance courante de la tâche fautive comme une tâche apériodique. Cela nous a amené à proposer une technique de prise en charge des tâches apériodiques. Cela consiste en l'exécution des tâches apériodiques en arrière plan pendant les temps d'inactivité des processeurs. Le test d'acceptation que nous avons proposé est de complexité en $O(k)$, où k est le nombre de tâches apériodiques présentes dans le système. Ce test exploite la répartition équitable des temps d'inactivité des processeurs ce qui permet de connaitre à deux unités de temps près le nombre de temps creux disponibles sur un intervalle de temps donné.
Nous avons ensuite considéré le problème de l'évolution dans le cadre plus général où l'application comporte des tâches pouvant être liées par une relation de précédence ou de partage de ressources. Nous utilisons dans ce contexte une stratégie d'ordonnancement à base de réseaux de Petri. Cette approche est la mieux adaptée pour ce modèle de tâches. Nous avons proposé des critères de sélection de séquences d'ordonnancement minimisant le temps de réponse et la gigue des tâches périodiques, et où les temps creux sont équitablement répartis. Là aussi nous utilisons le même test d'acceptation de tâches apériodiques proposé précédemment car le contexte s'y prête.

Nous avons en amont étudié le problème de l'ordonnancement temps-réel multiprocesseur. Nous avons particulièrement étudié le problème de la cyclicité en multiprocesseur, les ordonnancements P-équitables et les ordonnancements hors-ligne à base de

modèle.
Concernant la cyclicité, nous avons introduit la notion d'ordonnancement monotone et nous avons caractérisé l'entrée dans le régime permanent pour les ordonnancements monotones déterministes. Nous avons montré que les ordonnancements à priorités fixes, les ordonnancements à priorités dynamiques tels que EDF et LLF, et enfin les ordonnancements P-équitables tels que PF et PD^2 sont des ordonnancements monotones. Les simulations que nous avons effectuées nous ont permis de conjecturer que la date d'entrée dans le cycle est majorée par

$$max\{r_i\}_{i=1\ldots n} + T$$

dans le cadre des ordonnancement à priorités fixes et des ordonnancements P-équitables.
Concernant l'étude des ordonnancements P-équitables, nous avons étendu le principe de l'ordonnancement équitable à tout système de tâches périodiques. Nous avons donné une condition suffisante d'ordonnançabilité et nous avons montré qu'il existe un ordonnancement P-équitable sur m processeurs pour tout système de tâches périodiques indépendantes τ vérifiant

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq m.$$

Enfin, concernant les ordonnancements hors-ligne à base de réseaux de Petri, nous avons étendu les travaux de thèse de [60] et nous avons présenté des techniques d'extraction de séquences d'ordonnancement minimisant le temps de réponse et la gigue d'activation des tâches et dans lesquelles les temps creux sont équitablement répartis.
Il reste néanmoins beaucoup de points qui peuvent faire l'objet d'études futures. Ces points sont présentés dans les perspectives.

7.2 Perspectives

Les résultats d'impossibilité et les limites théoriques sont sans doute les conséquences de la complexité de la théorie de l'ordonnancements temps-réel multiprocesseur. Les travaux qui se sont intéressés à l'ordonnancement dans ce contexte ont utilisé

des hypothèses fortes sur les modèles de tâches ou sur le modèle d'exécution des tâches. Dans notre étude nous avons montré qu'on peut se défaire de certaines hypothèses et ne s'en tenir qu'à des modèles de tâches classiques. Mais il reste beaucoup à faire à notre point de vue au regard des restrictions faites dans nos travaux. Par exemple :

– nous avons supposé les tâches indépendantes pour l'étude de la cyclicité ;
– nous avons considéré que les tâches apériodiques qui arrivent dans le système ne doivent pas interagir avec les tâches périodique initiales ;
– etc... .

Nous présentons dans la section suivante nos travaux en cours pour nous défaire de ces hypothèses ainsi que nos perspectives.

7.2.1 Les travaux en cours

Nos travaux en cours se subdivisent en trois grands points : l'intégration de l'étude menée sur l'ordonnancement à base de réseaux de Petri dans l'outils PeNSMARTS, l'étude de la prise en charge de nouvelles tâches périodiques dans le système et enfin l'étude du fonctionnement multimodal des applications.

7.2.1.1 Intégration de nos travaux dans PeNSMARTS

Rappelons que PeNSMARTS (Petri Nets Scheduling, Modeling and Analysis of Real-Time Systems) est un outil de modélisation et de validation d'applications temps-réel à base de réseau de Petri. Mais nous avons vu que dans sa forme actuelle, PeNSMARTS ne prend en compte que des applications fonctionnant sur une plateforme monoprocesseur. Les études que nous menons ont pour objectif d'étendre les fonctionnalités de PeNSMARTS aux plateformes multiprocesseurs.

L'IHM de PeNSMART a été développé avec la version 4.5 TCL/TK mais une fois compilée, l'utilisation est indépendante de la version de tcl/tk (l'appel à wish est indépendant de la version). Le noyau a été développé en C/C++. Deux "parseurs" ont aussi été utilisés pour la reconnaissance des tâches et des contraintes. Ces "parseurs" ont été développés avec flex et bison.

Nous avons travaillé avec un étudiant de master à étendre le fonctionnement de PeNS-MARTS pour une plateforme à m processeurs avec $m \geq 1$. Nous sommes passés à la version 8.5 de TCL/TK pour le développement de l'IHM et nous avons utilisé flex et bison++ pour recréer les "parseurs". Dans le noyau nous avons supprimé l'utilisation de la tâche creuse. Nous avons ensuite résolu le problème de la charge processeur ainsi que celui de la tâche oisive. Rappelons que lorsqu'on passe en multiprocesseur, il peut avoir plusieurs tâches oisives. Nous avons enfin intégré les critères qui permettront de sélectionner les séquences d'ordonnancement souhaitées.
L'étude actuellement en cours concerne le contrôle de la profondeur du graphe des marquages accessibles et la prise en compte de plusieurs étiquettes (rappelons que les arcs du graphe des marquages ont plusieurs étiquettes qui correspondent aux tâches qui ont été ordonnancées à un instant donné).

7.2.1.2 Intégration de nouvelles tâches périodiques dans le système

Nous nous intéressons à la prise en charge de nouvelles tâches périodiques dans le cas où l'application comporte des tâches pouvant être liées par une relation de précédence ou de partage de ressources. Nous utilisons une stratégie d'ordonnancement à base de réseaux de Petri. Dans ce contexte d'ordonnancement, il y a deux façons de prendre en charge une nouvelle tâche dans le système :

1. recalculer l'ordonnancement à fournir au séquenceur avec les nouvelles tâche ou,
2. sélectionner des séquences permettant l'ajout de nouvelles tâches.

La première solution étant coûteuse, nous choisissons la deuxième avec pour le moment la restriction suivante : *une nouvelle tâche n'est prise en charge par le système que si elle est indépendante vis à vis des tâches déjà présentes dans le système*.
Les nouvelles tâches qui arrivent dans le système s'exécutent en arrière plan pendant les temps d'inactivité des processeurs. Nous distinguons les tâches périodiques initiales des nouvelles tâches périodiques qui arrivent dans le système. Pour cela nous appelons *partie statique du système* les tâches périodiques initiales et *partie dynamique du système* les tâches apériodiques et les nouvelles tâches périodiques.

La partie statique est exécutée selon un plan statique calculé hors-ligne. Nous utilisons un ordonnancement à base de réseaux de Petri et nous mettons en œuvre les techniques de sélection d'ordonnancement établies pendant notre étude. Les tâches de la partie dynamique s'exécutent en arrière plan. Elles sont pris en charge par des serveurs qui les ordonnancent pendant les temps d'inactivité des processeurs. L'ordonnancement des tâches de la partie dynamique du système est appelé dans la suite *ordonnancement dynamique*. Les serveurs sont définis de la façon suivante :
soit τ un système de n tâches périodiques avec $U_\tau < m$, où m est le nombre de processeurs de la plateforme. Rappelons que si $m - U_\tau = k_0 + u_0$ avec $k_0 \in \mathbb{N}$ et $0 \leq u_0 < 1$, alors à chaque instant t, il y a au moins k_0 temps creux et au plus $k_0 + 1$ temps creux globalement dans le système (cela s'explique par le fait que les temps creux sont équitablement répartis dans le plan d'exécution des tâches de la partie statique). Nous pouvons donc mettre en œuvre un serveur de tâches apériodiques σ_A et k_0 serveurs de tâches périodiques σ_P avec $\sigma_P = \left\{\sigma_{p_1}, \ldots, \sigma_{p_{k_0}}\right\}$.

7.2.1.2.1 L'ordonnancement du système : Comme $|\sigma_P| = k_0$, alors il n'y a pas de concurrence entre les serveurs $\sigma_{p_i},\ i = 1 \ldots k_0$. À chaque instant, chaque serveur aura sa part de temps processeur. Pour tenir compte de l'urgence des tâches dynamique, nous utilisons l'algorithme d'ordonnancement *EDF* pour les départager. Elles sont affectées aux serveurs selon une politique bin-packing. L'algorithme que nous utilisons est l'algorithme $Worst - Fit(WF)$ [40] pour être sûr que tous les serveurs auront des tâches à traiter .
Les tâches apériodiques sont prises en charge par le serveur d'apériodique σ_A. Le test d'acception est celui mis en œuvre dans notre étude.
L'architecture globale du système est illustrée par la figure 7.1. Chaque serveur σ_{p_i} a son ordonnanceur local. A chaque instant, il y a au plus k_0 nouvelles tâches périodique prêtes à être exécutées, au plus une tâche apériodique et enfin $m - k_0 - 1$ tâches périodiques initiales.

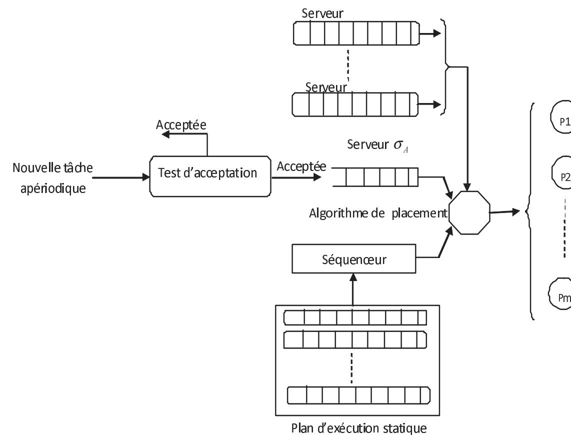


Figure 7.1 – Ordonnancement des tâches apériodiques et nouvelles tâches périodiques :

7.2.1.3 Etude du fonctionnement multimodal des applications

Nous parlons de fonctionnement multimodal quand le système est capable de passer d'une configuration à une autre. Une configuration est un ensemble de tâches défini hors-ligne. Le système a donc plusieurs spécifications fonctionnelles ou une série de fonctions. Chaque fonction est effectuée par un ensemble de tâches liées par des contraintes de ressources, de précédence et de temps. Chaque fonction est caractérisée par son coût d'exécution et les ressources partagées. Un tel fonctionnement repose sur deux mécanismes :

- un mécanisme de détermination de la configuration la mieux adapté à une évolution donnée ;
- un mécanisme de changement de configuration.

Peu de travaux traitant de ce mode de fonctionnement existent dans la littérature [21, 47, 57]. Dans leurs travaux,[21, 57] considèrent une application comme un ensemble d'entités. Chaque entité est composée d'un ensemble de tâches qui constituent plusieurs configurations. Quand une entité est activée, en fonction de l'état courant du système, la configuration la plus adaptée pour l'entité est choisie.

Les travaux de [47] présentent une approche similaire. Plusieurs configuration du système sont définies hors-ligne. Le concepteur fournit alors un automate pour passer d'une

configuration à une autre et c'est un *régisseur* qui décide de changer l'état de l'automate. Nous adoptons une approche semblable pour le fonctionnement multimodale. Nous appelons mode de fonctionnement l'exécution d'une configuration donnée. Dans notre étude nous ajoutons une configuration intermédiaire validée hors ligne qui prend en charge le passage d'une configuration à une autre, de sorte que la transition ne laisse pas le système dans un état incohérent. Nous appellerons par la suite, l'exécution de cette configuration intermédiaire, *le mode de survie*. Le mode de survie est en réalité un fonctionnement nominal du système. Ce mode de survie peut être aussi utilisé pour d'autres opérations intermédiaires comme par exemple le chargement des panneaux solaire d'un satellite. La configuration intermédiaire sera composée de tâches essentielles à ce fonctionnement nominal. Les tâches de cette configuration peuvent donc constituer le *système de base* dont nous avons parlé dans le chapitre précédent. Pour l'analyse du système, la configuration des tâches du système de base notée τ est modélisée par un réseau de Petri autonome fonctionnant sous la règle du tir maximal et validé hors-ligne. Ce système de base est composé de tâches périodiques à départs simultanés pour faciliter son étude. En effet cela limite la profondeur du graphe des marquages accessibles à générer et le nombre limité des tâches de ce système de base limite aussi la taille du graphe. Un plan statique d'exécution optimisé du système de base est alors fourni.

Chaque mode de fonctionnement est constitué d'un ensemble de tâches prises en charge par des serveurs. Nous mettons en œuvre un serveur d'apériodiques σ_A, et des serveurs de tâche périodiques σ_P. Les tâches prises en charge par les serveurs peuvent partager des ressources critiques.

7.2.1.3.1 Le mécanisme de choix d'une configuration : Lorsqu'un évènement extérieur à l'application nécessite de modifier le comportement du système, celui ci passe en *mode de survie* afin de changer de configuration. L'approche que nous avons choisie afin de réaliser ce mécanisme consiste à utiliser une variable représentative du mode fonctionnement courant. Changer de mode consiste alors à mettre à jour cette variable et à réaliser certaines actions telles que le choix d'une configuration existante ou le téléchargement d'une nouvelle configuration. On peut alors mettre en œuvre une tâche

indépendante qui scrute l'état du système et qui peut positionner cette variable. Cette tâche sera appelée par la suite *tâche de scrutation* et la variable appelée *variable d'état*. La variable d'état est accédée en lecture par l'ensemble des tâches du système et en écriture par cette la *tâche de scrutation*. Pour donner la possibilité à un opérateur externe de demander un changement de mode, on donne alors la possibilité à un opérateur externe de positionner cette variable d'état et de télécharger une nouvelle configuration. La variable d'état qui est de fait globale doit donc être protégée par un mécanisme de verrou en lecture/écriture. Ainsi, pendant la modification de la variable, on ne peut ni lire, ni modifier cette variable.

7.2.1.3.2 Le mécanisme de changement de configuration : Pour garantir une cohérence lors du changement mode, une configuration ayant commencé dans un mode doit terminer dans ce mode. Cela veut dire que dès que la variable d'état a été positionnée pour un nouveau mode, toutes les instances courantes des tâches de l'ancienne configuration doivent terminer avant qu'une nouvelle configuration ne soit activée.
Si un changement de mode de fonctionnement est demandé à un instant t, le temps pris pour passer du mode en cours au *mode de survie* est le maximum des temps réponse des instance en cours. On peut borner ce temps en prenant le temps de réponse maximal de toutes les tâches de la configuration. Si nous notons D_{CM} de cette durée , qui est la durée maximale d'un changement de mode, alors

$$D_{CM} = max\left\{ \left\lceil \frac{C_i + 1}{U_{\sigma_{P_j}}} \right\rceil + B_i | (\tau_i \in \tau_{\sigma_{p_j}}) \wedge (\tau\sigma_{p_j} \in \sigma_P) \right\}.$$

$\tau_{\sigma_{p_j}}$ désigne l'ensemble des tâches prises en charge par le serveur σ_{p_j} avec $\sigma_{p_j} \in \sigma_P$. B_i désigne le temps de blocage d'une tâche τ_i. Ainsi donc, si un changement de mode a été initié à l'instant t, alors aucun autre changement de mode ne peut être demandé avant $t + D_{CM}$.

7.2.1.3.3 Architecture globale : L'architecture globale de l'approche que nous proposons pour le fonctionnement multimodal est illustrée sur la figure 7.2. Le système de

tâches est composé des tâches du système de base qui permettent la transition d'une configuration à une autre et des tâches de la sous-partie partie dynamique. La sous-partie dynamique est composée de tâches apériodiques et de tâches périodiques prises en charge par des serveurs.
La sous partie statique ou système de base est exécutée selon un plan statique fourni à un séquenceur. Chaque serveur de la sous partie statique a son propre ordonnanceur local. Les serveurs mettent en œuvre une politique d'ordonnancement EDF.

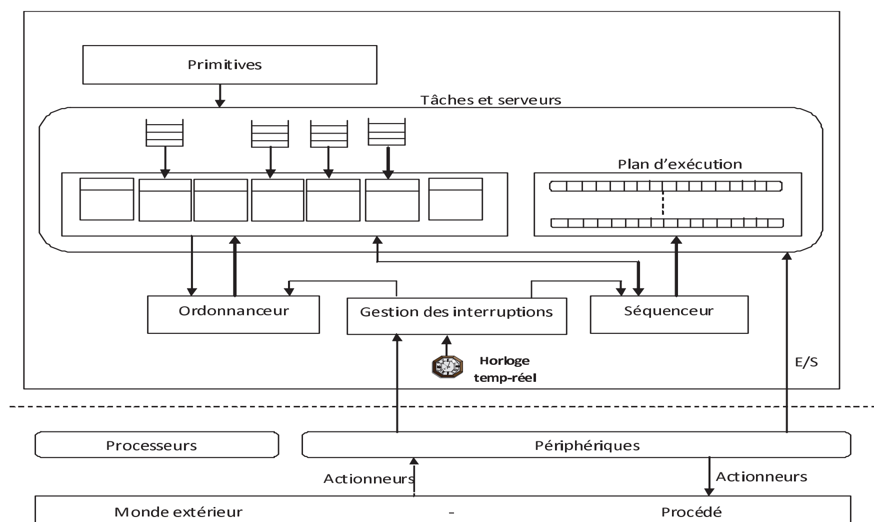


Figure 7.2 – Illustration de l'architecture globale de l'approche

7.2.2 Études à venir

Le travail en cours ne peut aboutir que si on se défait de l'hypothèse “tâches indépendantes” pour la prise en charge des tâches apériodiques et des nouvelles tâches périodiques.
Dans nos études à venir, nous allons nous défaire de cette hypothèse afin de considérer des modèles de tâches plus réalistes.
Ensuite au cours de l'étude de la panne processeur, nous n'avons considéré que le cas de la panne d'un seul processeur. On pourrait étendre cette étude au cas où plusieurs processeurs tombent en panne et envisager aussi une reprise partielle des tâches fautives.

Enfin, si l’étude du fonctionnement multimodal est bien élaborée, elle pourrait s’intégrer dans un projet d’expérimentation sur une plateforme linux temps-réel selon une approche ’dual-kernel’.

LISTE DES TABLEAUX

1.I Récapitulatif de la complexité des algorithmes multiprocesseur partitionnés. n est le nombre de tâches et m le nombre de processeurs. 30

2.I Système de tâche $\tau = \{\tau_1(0,4,8,8); \tau_2(0,5,8,8); \tau_3(0,4,8,8)\}$ 52

3.I Date d'entrée dans le cycle sur deux processeurs et selon les algorithmes RM, EDF et LLF 90

3.II Résultats de la simulation sur la cyclicité. L'algorithme d'ordonnancement utilisé est PF. t_0 désigne le temps d'entrée dans le cycle. 90

5.I Résultats de la simulation sur l'ordonnançabilité. N désigne le pourcentage de systèmes ordonnançable sur l'échantillon considéré. 118

6.I Table des paramètres dynamiques des tâches apériodiques 130

LISTE DES FIGURES

1 Système temps-réel strict : 2
2 Système temps-réel “mou” : 2
3 Système temps-réel “ferme” : 3
4 Système temps-réel incrémental : 3
5 Système temps-réel : exécutif et tâches temps-réel. 5
6 Système de tâches : 8
7 Ordonnancement global 11
8 Ordonnancement partitionné 11

1.1 Activations périodiques de période T_i 22
1.2 Activations sporadiques 22
1.3 Ordonnancement valide 26
1.4 Ordonnancement non valide : 26
1.5 Anomalie d’ordonnancement : 27
1.6 Anomalie d’ordonnancement : 27
1.7 Anomalie d’ordonnancement : 27
1.8 Instance en cours d’une tâche : temps processeur reçu par l’instance en cours et temps processeur restant à exécuter par cette même instance. 32
1.9 P-équité 36
1.10 Ordonnancement P-équitable de la tâche $\tau_i = (0,3,5,5)$. 38

2.1 Modélisation d’une application : 51
2.2 Modélisation de l’horloge de la tâche τ_i 52
2.3 Décomposition en bloc de la tâche τ_i 54
2.4 Contrainte de successeur 59

3.1 Temps creux acyclique : 69
3.2 Début du régime permanent : 70
3.3 Temps creux complet sur 2 processeurs 73

3.4 Temps creux partiel sur 2 processeurs 73
3.5 La cyclicité sous *EDF* : 89
3.6 La cyclicité sous *LLF* : 89

4.1 Cas le pire d'occurrence d'une panne 95
4.2 Paramètres d'une tâche : échéance principale et échéance de reprise . . 96
4.3 Paramètres décalés d'une tâche dans l'optique d'une reprise totale . . . 96
4.4 Ordonnancement EDF d'un système de tâches sur 2 processeurs. 97
4.5 Ordonnancement EDF d'un système de tâches sur 3 processeurs. 97
4.6 Panne processeur en fin d'instance d'une tâche 98
4.7 Ordonnancement EDF d'un système de tâches sur 2 processeurs avec partage d'une ressource critique. 102
4.8 Ordonnancement EDF d'un système de tâches sur 3 processeurs avec partage d'une ressource critique. 102
4.9 Ordonnancement EDF d'un système de tâches sur 2 processeurs. 103
4.10 Ordonnancement EDF d'un système de tâches sur 3 processeurs. 103
4.11 Ordonnancement EDF d'un système de tâches sur 3 processeurs avec une panne d'un processeur à l'instant $t_p = 3$. 104
4.12 Ordonnancement EDF d'un système de tâches sur 3 processeurs avec une panne d'un processeur à l'instant $t_p = 3$. 104

5.1 Ordonnancement P-équitable de la tâche $\tau_i = (2,3,5,8)$ 110
5.2 Exemple de graphe : 112
5.3 Graphe 2 : 119
5.4 Graphe 3 : 119
5.5 Ordonnancement par PF du système τ sur 3 processeurs 121
5.6 Ordonnancement par PF du système $\tau^{'}$ sur 3 processeurs 121

6.1 Ordonnancement des tâches apériodiques : méthode conjointe 124
6.2 Ordonnancement du système de tâche τ sans la tâche oisive 127
6.3 Ordonnancement du système de tâche τ avec la tâche oisive 127

6.4 Comparaison de notre méthode à la méthode conjointe 132
6.5 Tâche oisive avec parallélisation 136
6.6 Tâche oisive sans parallélisation 136
6.7 Répartition des temps creux : 137

7.1 Ordonnancement des tâches apériodiques et nouvelles tâches périodiques : 147
7.2 Illustration de l'architecture globale de l'approche 150

BIBLIOGRAPHIE

[1] R. Alur, C. Courcoubetis et D. Dill. Model-checking in dense real-time. *Information and Computation*, 101 (1):2–34, 1993.

[2] R. Alur et D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[3] B. Anderson et J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. Dans *Proceedings of the* 15th *Euromicro Conference on Real-Time Systems*, pages 33–40, 2003.

[4] J. Anderson, A. Block et A. Srinivasan. Pfair scheduling : Beyond periodic task sytems. Dans *Proceedings of the* 12th *Euromicro Conference on Real-Time Systems*, pages 35–43. Chapman and Hall, 2000.

[5] J. Anderson et A. Srinivasan. Early-release fair scheduling. Dans *Proceedings of the* 12th *Euromicro Conference on Real-Time Systems*, pages 35–43. Chapman and Hall, 2000.

[6] J. Anderson et A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. Dans *Proceedings of the* 13th *Euromicro Conference on Real-Time Systems*, pages 76–85, 2001.

[7] J. Anderson et A. Srinivasan. A new look at pfair priorities. *Real-time Systems Journal*, Oct 1999.

[8] B. Andersson. *Static Priority Scheduling on Multiprocessors*. Thèse de doctorat, Department of Comp. Eng., Chalmers University, 2003.

[9] J. Andersson, S. Baruah et J. Jonsson. Static priority scheduling on multiprocessors. Dans *Proceedings of the* $22^{n}d$ *IEEE RealTime Systems Symposium*, pages 193–202, 2001.

[10] D. Aoun, A.M. Déplanche et Y. Trinquet. Pfair scheduling improvement to reduce interprocessor migrations. Dans *Proceedings of RTNS 08*, pages 131–138, 2008.

[11] ARG. Ravenscar profile for high-integrity systems. Rapport technique, Technical report, ISO/IEC/JTC1/SC22/WG9, 2003.

[12] T.P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. Dans *Proceedings of the* $24^t h$ *IEEE Real-Time Systems Symposium*, 2003.

[13] T.P. Baker. An analysis of edf schedulability on a multiprocessor. *Transactions on Parallel and Distributed Systems*, 16(8):760–768, 2005.

[14] T.P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *The Journal of Real-Time Systems*, 32:49–71, 2006.

[15] M. Barabanov. *A linux-based real-time operating system.* Master's Thesis, New Mexico Institute of Mining and Technology, june 1997.

[16] S. Baruah. Fairness in periodic real-time scheduling. Dans *Proceedings of the* $16^t h$ *IEEE Real-Time Systems Symposium*, pages 200–209, 1995.

[17] S. Baruah, J. Gehrke et C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. Dans *Proceedings of the* 9th *International Parallel Processing Symposium*, pages 280–288, April 1995.

[18] S.K. Baruah, N.K. Cohen, C.G. Plaxton et D.A. Varvel. Proportionate progress : a notion of fairness in resource allocation. *Algorithmica*, 15:600–625, 1996.

[19] S.K. Baruah et J. Goossens. Scheduling real-time tasks : Algorithms and complexity. *In Handbook of scheduling :Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (ed). Chapman Hall/CRC Press*, 2004.

[20] S.K. Baruah, L.E. Rosier et R.R. Howell. Algorithms and complexity concernig the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, pages 301–324, 1990.

[21] A. Bondavalli, J. Stankovic et L. Strigini. Adaptable fault tolerance for real-time systems. Rapport technique, Technical Report UM-CS-1994-039, University of Massachusetts, Amherst, Computer Science, May 1994.

[22] A. Burchard, J. Liebeherr, Y. Oh et S.H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44 (12), Dec. 1995.

[23] A. Burchard, Y. Oh, J. Liebeherr et S.H. Son. A linear time online task assignment scheme for multiprocessor systems. Dans *IEEE Workshop on Real-Time Operating Systems and Software*, May 1994.

[24] A. Burns. The ravenscar profile. Rapport technique, Technical Report, University of York., 2002.

[25] A. Burns, R. Davis et S. Punnekkat. Feasibility analysis of fault-tolerant real-time tasks sets. Dans *Proc. of the 8th Euromicro Workshop on Real-Time System*, 1996.

[26] A. Burns et A.J. Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley,second edition, 1996.

[27] G.C. Buttazzo. *Hard Real-Time Computing Systems,predictable scheduling,algorithms and applications*. Kluwer Academic Publishers, 1997.

[28] C.Han et H. Tyan. A better polynomial-time scheduling schedulability test for real-time fixed-priority scheduling algorithm. Dans *Real-Time Systems Symposium*, pages 36–45, 1997.

[29] E. Chanthery et Y. Pencolé. Modélisation et intégration du diagnostic actif dans une architecture embarquée. Dans *Journal Européen des Systmes Automatisés, MSR 2009*, pages 789–803, 2009.

[30] H. Chetto et M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on software engineering*, 15(10):1261–1269, 1989.

[31] H. Chetto, M. Silly et T. Bouchentouf. Dynamic scheduling of real time tasks under precedence constraints. *The Journal of the Real time Systems*, 2:115–127, 1990.

[32] A. Choquet-Geniet. Panorama de l'ordonnancement temps-réel. Dans *Actes de l'école d'éte E.T.R'03*, pages 213–226, 2003.

[33] A. Choquet-Geniet. Un premier pas vers l'étude de la cyclicité en environnement multiprocesseurs. *Actes de la conférences RTS'05*, pages 289–302, 2005.

[34] A. Choquet-Geniet. *Les réseaux de Petri - Un outil de modélisation*. Dunod, 2006.

[35] A. Choquet-Geniet. *Systèmes parallèles et Temps-réel*. Mémoire d'Habilitation à Diriger des Recherches, Univ. Poitiers, Decembre 2000.

[36] A. Choquet-Geniet, D. Geniet et F. Cottet. Exhaustive computation of the scheduled task execution sequences of a real-time application. Dans *Proc. of FTRTFT'96*, October 1996.

[37] A. Choquet-Geniet et E. Grolleau. Minimal schedulability interval for real time systems of periodic tasks with offsets. *Theoretical of Computer Science*, pages 117–134, 2004.

[38] A. Choquet-Geniet et S. Malo. Finding cyclicity behavior in multiprocessor scheduling. Rapport technique, LISI - ENSMA and University of Poitiers, 2009.

[39] E.M. Clarke, E.A. Emerson et A.P. Sisla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. Prog. Lang. Syst.*, 8(2):244–263, 1986.

[40] E.G Coffman, G. Galambos, S. Martello et D. Vigo. Bin packing approximation algorithms : Combinational analysis. 1998.

[41] E.G. Coffman, M.R. Garey et D.S. Johnson. Approximation algorithms for bin-packing-a survey,. *Ananlysis and Design of Algorithms in Combinatorial Optimization, G. Ausiello and M. Lucertini*, 1981.

[42] L. Cucu et J. Goossens. Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors. Dans *The* 11th *IEEE International Conference on Emerging Technologies and Factory Automation*, pages 397–405. IEEE Computer Society Press, 2006.

[43] L. Cucu et J. Goossens. Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems. Dans *Design, Automation and Test in Europe*, pages 1635–1640. IEEE Computer Society, 2007.

[44] S. Davari et S.K. Dhall. On a periodic real-time task allocation problem. *Annual International Conference on System Sciences*, 1986.

[45] L. David, F.Cottet et E.Grolleau. Gigue temporelle et ordonnancement par échéance dans les appilcations temps-réel. *In Pro. of IEEE Conférence Internationale Francophone d'Automatique (CIFA 2000), Lille, France*, Juillet 2000.

[46] D. Decotigny. *Une infrastructure de simulation modulaire pour l'évaluation de performances de systèmes temps-réel*. Thèse de doctorat, Université de Rennes 1, France, 7 Avril 2003.

[47] J. Delacroix. Stabilitée et régisseur d'ordonnancement en temps réel. *Technique et Sciences Informatiques*, 13(2):223–250, 1994.

[48] M.L. Dertouzos et A.K.L. Mok. Multiprocessor scheduling in hard real-time environment. *IEEE transactions on sofware Engineering*, 15(12):1497–1506, 1989.

[49] S.K. Dhall. Scheduling periodic-time critical jobs on single processor and multiprocessor computing systems. *PhD Thesis*, April 1977.

[50] S.K. Dhall et C.L. Liu. On a real-time scheduling problem. *Operation Research*, (26(1)):127–140, 1978.

[51] W. Dinkel, M. Frisbie et J. Woltersdorf. *Kurt-Linux User Manuel*. UC Kansas at Lawrence, Mars 2002.

[52] J. Leung Ed. *Handbook of scheduling :Algorithms, Models, and Performance Analysis*. Chapman and Hall/ CRC Press, Washington, D.C., 2004.

[53] C. Ekelin et J. Jonsson. Solving embedded system scheduling problems using constraint programming. Dans *IN IEEE RTSS*, 2000.

[54] G. Florin et S. Natkin. Les réseaux de petri stochastiques. *Techniques et Sciences Informatiques*, 4(1):143–160, February 1985.

[55] IEEE Standard for Information Technology. Portable operating system interface 1003.1b and 1003.1c. Rapport technique, IEEE, 1993.

[56] M.R. Garey et D.S. Johnson. Schedulling tasks with nonuniform deadlines on two processors. *Journal of the Association for Computing Machinery*, 23(3):461–467, 1976.

[57] O. Gonzalez, H. Shrikumar, J.A. Stankovic et K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time constraints. Dans *In Proceedings of the 18th IEEE Real-Time Systems Symposium, San Francisco, California. U. Mass*, Dec. 1997.

[58] V. Grassi, L. Donatiello et S. Tucci. On the optimal checkpointing of critical tasks and transaction-oriented systems. *IEEE Transactions on Software Engineering*, 18(1):72–77, January 1992.

[59] E. Grolleau. Projet pensmarts. Dans *Actes de ETR'97*, 1997.

[60] E. Grolleau. *Ordonnancement Temps-Réel Hors-Ligne Optimal à l'Aide de Réseaux de Petri en Environnement Monoprocesseur et Multiprocesseur*. Thèse de doctorat, Univerité de Poitiers, 1999.

[61] E. Grolleau et A. Choquet-Geniet. Scheduling real-time systems by means of petri nets. Dans *Proc. of* $25^t h$ *Workshop on Real-Time Programming*, pages 95–100. Universidad Politécnica de Valencia, 2000.

[62] E. Grolleau et A. Choquet-Geniet. Real-time scheduling in multiprocessor environment by means of Petri nets. *Proceedings of RTS 2001*, pages 189–206, 2001.

[63] E. Grolleau et A. Choquet-Geniet. Off line computation of real time schedules by means of petri nets. *Journal of Discrete Event Dynamic Systems*, 12:311–333, 2002.

[64] E. Grolleau, A. Choquet-Geniet et F. Cottet. Modélisation des systèmes réactifs par réseaux de petri autonomes en vue de leur analyse hors-ligne. Dans *Proceedings of MSR'99*, pages 17–26. Hermès, 1999.

[65] G. H. Kopetz et al. The programmer's view of mars. *Proceedings of the Real-Time Systems Symposium*, pages 223–226, 1992.

[66] A.C. Heursch. Preemption concepts, rhealstone benchmark and scheduler analysis of linux 2.4. *In proceedings of the Real-Time and Embedded Computing Expo and Conference*, 2001.

[67] K.S. Hong et J.Y Leung. On-line scheduling of real-time tasks. *IEEE Computer Society*, 1988.

[68] R. Janicki, P.E. Lauer, , M. Koutny et R. Devillers. Concurrent and maximally concurrent evolution of non sequential systems. *Theorical Computer Science*, 43(2-3):213–238, 1986.

[69] K. Jensen. *Coloured Petri nets, basic concepts, analysis methods and pratical use*. Monographs in theoretical Computer Science. Springer Verlag, 1996.

[70] M. Joseph et P. Pandya. Finding response times in a real-time system. *The computer journal*, 29(5):390–395, 1986.

[71] L.R. Ford Jr et D.R. Fulkerson. *Flows in networks*. Princeton University Press, 1962.

[72] R.M. Karp et V. Ramchandani. Parallel algorithms for shared-memory machines. Dans J.V. Leuwen, éditeur, *Algorithms and complexity*, pages 869–935. MIT press, 1990.

[73] C.M. Krishna, K.G. Shin et Y.-H. Lee. Optimization criteria for checkpointing. *Communication of the ACM*, 27(10):1008–1012, October 1984.

[74] S. Lauzac, R. Melhem et D. Mosse. An improved rate-monotonic admission control and application. *IEEE Transactions on Computers*, 52(3), March 2003.

[75] S. Lauzac, R. Melhem et D. Mosse. Comparaison fo global and partitioning schemes for scheduling rm tasks on multiprocessor. Dans *Euromicro Workshop on Real-Time Systems*, June 1998.

[76] P.A. Lee et T. Anderson. Fault tolerance : Principe and practice. *Springer-Verlag, second edition*, 1990.

[77] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. Dans *Proc. of the 11th IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[78] J. Leung et M. Merill. A note on preemptive scheduling of periodic real-time tasks. *Information processing letters,*, 11(3):115–118, 1980.

[79] J. Leung et J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, pages 237–250, 1982.

[80] J.Y.T. Leung et M.L. Merill. A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.

[81] C.L. Liu et J.W. Layland. Scheduling algorithms for multiprogramming in a hard real- time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[82] J. Liu et R. Ha. Efficient methods of validating timing constraints. *Advances in Real-Time Systems*, pages 199–223, 1995.

[83] J.W.S Liu. *Real-Time Systems*. Prentice Hall, 2000.

[84] J.M. Lopez, J.L. Diaz et D.F. Garcia. Minimum and maximum utilization bounds for multiprocessor rate monotonic scheduling. *IEEE Trans. on Parallel and Distributed Systems*, 15(7), Jul. 2004.

[85] J.M. Lopez, M. Garcia, J.L. Diaz et D.F. Garcia. Worst-case utilization bound for edf scheduling on real- time multiprocessor systems. *Euromicro Workshop on Real Time Systems*, pages 25–33, 2000.

[86] J.M. Lopez, M. Garcia, J.L. Diaz et D.F. Garcia. Utilization bounds for multiprocessor rate-monotonic scheduling. *Real Time Systems*, 24(1), Jan. 2003.

[87] J.M. Lopez, M. Garcia, J.L. Diaz et D.F. Garcia. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real Time Systems*, 28(1), Oct. 2004.

[88] C. Macq et J. Goossens. Limitation of the hyper-period in real-time periodic task set generation. Dans Teknea, éditeur, *Proceedings of the 9th international conference on real-time systems*, pages 133–148, Paris France, March 2001. ISBN 2-87717-078-0.

[89] G. Miremadi et J. Torin. Evaluating processor-behaviour and three error-detection machanisms using physical fault-injection. *IEEE Transactions on Reliability*, 44 (3):441–453, September 1995.

[90] A.K. Mok et M.L. Dertouzos. Multi processor scheduling in a hard real-time environment. Dans *Proc. of* 7th *Texas Conference on Computer Systems*, 1978.

[91] R.R Muntz et E.G. Coffman jr. Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the association for computing Machinery*, 17(2): 324–338, 1970.

[92] D.I. Oh et T. Baker. Utilization bound for n-processor rate monotonic scheduling with static processor assignment. *Real-Time Systems.*, 15(2):183–192, 1998.

[93] Y. Oh et S.H. Son. Fixed priority scheduling of periodic tasks on multiprocessor systems. Rapport technique, Technical report CS-95-16. Univ. Of Virginia. Dept. of Computer Science, March 1995.

[94] Y. Oh et S.H. Son. Tight performance bounds of heuristics for real-time scheduling problem. Rapport technique, Technical report CS-93-24. Univ. Of Virginia. Dept. of Computer Science, May 1993.

[95] G. Cabilic A. Colin D. Decotigny P. Chevochot, I.Puaut et M. Banâtre. Hades : a distributed system for dependable hard real- time applications built from cots components. Rapport technique, IRISA, October 2000.

[96] S. Punnekkat. *Schedulability Analysis for Fault Tolerant Real-Time Systems*. Thèse de doctorat, University of York, 1997.

[97] K. Ramamritham, J.A. Stankovic et P. Shiah. O(n) scheduling algorithms for real-time multiprocessor systems. *International Conference on Parallel Processing*, 3: 143–153, 1989.

[98] S. Ramamurthy. Scheduling periodic hard real-time tasks with arbitrary deadlines on multiprocessors. Dans *In proceedings of the 23rd Real-Time Systems Symposium*, page 59, 2002.

[99] S. Ramamurthy et M. Moir. Static-priority scheduling on multiporcesors. Dans *Proceedings of the* $21^{s}t$ *IEEE Real-Time Systems Symposium*, pages 69–78, 2000.

[100] C. Ramchandani. Analysis of asynchronous concurrent systems by petri nets. Rapport technique, MIT, project MAC, 1974.

[101] R.D. Schlichting et F.B. Schneider. Fail-stop processor : An approach to designing fault- tolerant computing sysytems. 1983.

[102] K.G. Shin, T. Lin et Y. Lee. Optimal checkpointing of real-time tasks. *IEEE Transactions on Computers*, C-36(11), 1987.

[103] P.G. Sorenson. A methodology for real-time system development. *PhD Thesis, University of Toronto*, Canada,1974.

[104] M. Spuri. Analysis of deadline scheduled real-time system. Rapport technique, Technical Report 2772,INRIA Rocquencourt, Jan 1996.

[105] A. Srinivasan et J. Anderson. Fair scheduling of dynamique task systems on multiprocessors. Dans *Journal of Systems and Software*, April 2003.

[106] A. Srinivasan et J. Anderson. Optima rate-based scheduling on multiprocessors. Dans *In Proceedings of the 34th ACM Symposium on Theory of Computing*, pages 189–198, May 2002.

[107] A. Srinivasan et S. Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information Processing Letters*, 84, 2002.

[108] A. Srinivasan, P. Holman et J. Anderson. Integrating aperiodic and recurrent tasks on fair- scheduled multiprocessors. Dans *14th Euromicro Conference on Real-Time Systems (ECRTS 2002),19-21 June 2002, Vienna, Austria, Proceedings*, pages 189–198. IEEE Computer Society, 2002.

[109] J.A. Stankovic, M. Spuri, M. Di Natale et G. Buttazzo. Implications of classical scheduling results for real time systems. *IEEE Computer*, 28(9):16–25, 1995.

[110] J.A. Stankovoc, K. Ramamritham, D. Niehaus, M.Humphrey et G.Wallace. The spring system : Integrated support for complex real-time systems. Rapport technique, Technical Report CS-98-18, Department of Computer Science, University of Virginia, August 1 1998.

[111] P. Starke. Some properties of timed petri nets under te earliest firing rule. Dans *Advances in Petri nets '90*, LNCS 424. Springer Verlag, 1990.

[112] K. Tindell, A. Burns et A.J. Wellings. Allocating real-time tasks (an np-hard problem made easy). *The Journal of Real Time Systems*, 4(2):145–165, 1992.

[113] H. Tokuda, T. Nakajima et P. Rao. Real-time mach :toward a predictable real-time system. Dans *In Proceedings of USENIX Mach Workshop*, pages 73–82, Octobre 1990.

[114] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung et C. Kintala. Checkpointing and its applications. *In Proc. 25th Fault-Tolerant Computing Symposium*, pages 22–31, June 1995.

[115] Y.C Wang et K.J. Lin. Providing real-time support in the linux kernel. Dans *In proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*, page 1, 1999.

[116] J. Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence and exclusion relations. *IEEE Transactions on Software Engineering*, 19 (2):139–153, 1993.

ABSTRACT

A real-time system is one whose logical correctness is based both on the correctness of the outputs and on their timeliness. It must satisfy explicit (bounded) response-time constraints or risks severe consequences, including system failure. Consequently, a key requirement for real-time systems is the end-to-end delay in task execution, a critical issue is the design and analysis of these time critical systems.

Now, systems are increasingly complex, using several processors to process a large number of tasks. Furthermore, the environment in which these systems are used is complex and evolving. For these reasons, we focus on the schedulabilty analysis of real-time applications on multiprocessor platforms and on the scalability of these applications. In this regard we propose a new approach for fault-tolerant scheduling on multiprocessors with global preemptive EDF and Pfair policy. Our approach guarantees the completion of a scheduled task before its deadline in the presence of a processor failure. It requires more results on schedulability theory on multiprocessors when complex tasks systems are considered. First, we address the cyclicity problem for global multiprocessor scheduling. We propose a large class of scheduling strategies, the monotonous class and show how to characterize the beginning of the steady state. Then we show that most of the classical algorithms belong to this class. Then, we extend the notion of Pfairness to the context of asynchronous tasks with contrained deadlines. We investigate feasibility conditions and we propose a rather efficient one. Finally, we consider the problem of aperiodic tasks with contrained deadline adjunction. We propose an acceptance protocol, which relies on a fair distribution of the idle time units. The accepted aperiodic tasks are then scheduled in background. Next, we consider interacting periodic tasks. We still try to distribute the idle time units fairly. For that aim, we consider a model-driven approach, based on the modelling of the application by a Petri net and we propose extraction rules which can accommodate our acceptance protocol.

Keywords: cyclicity of schedules, scalability, Pfair scheduling, optimal scheduling, Fault-tolerance, Petri nets, multimodal scheduling.

RÉSUMÉ

Les applications temps réel, le plus souvent dédiées au contrôle de procédé, sont soumises à des contraintes temporelles strictes, destinées à garantir la sécurité et la cohérence du procédé contrôlé. Les applications temps réel étant des applications multi tâches, elles doivent être ordonnancées, le critère sine qua non de qualité de la stratégie d'ordonnancement étant la garantie du respect des contraintes temporelles.
Ces applications sont de plus en plus souvent déployées sur des architectures multiprocesseurs. Le problème de l'ordonnancement doit donc être posé dans ce contexte, où de nombreux problèmes doivent encore être abordés. Notons tout d'abord que dans le cas multiprocesseur, il n'existe pas d'ordonnancement en ligne optimal dans le cas général, le problème de l'ordonnancement est NP-complet, et des anomalies d'ordonnancement apparaissent même lorsque l'on ne considère que des tâches indépendantes (une durée d'exécution plus courte que prévue peut provoquer une faute temporelle).
Nous avons envisagé de prendre en compte la possibilité que des pannes matérielles surviennent. Nous avons étudié les mécanismes de reprise après la panne d'un processeur, dans le cas où l'application est ordonnancée par EDF, puis par un algorithme P-équitable. Ces mécanismes nécessitent de disposer d'un certain nombre de résultats généraux. Tout d'abord, nous avons étendu la définition de la P-équité à un contexte plus large que celui de la littérature, à savoir aux tâches à départs différés et à échéances contraintes, puis nous avons établi une condition suffisante d'ordonnançabilité dans ce contexte, dont nous avons étudié les performances à l'aide de simulations. Enfin, nous avons proposé un protocole efficace de gestion des flux apériodiques qui s'appuie sur une distribution P-équitable des temps creux. Cette répartition peut être obtenue soit en ordonnançant les tâches périodiques par une stratégie P-équitable quand le contexte le permet, soit via une analyse hors-ligne à base de réseaux de Petri. Nous avons proposé une technique d'extraction qui permet de forcer la répartition équitable des temps creux.
Mot clés : cyclicité des ordonnancements, ordonnancements évolutifs, ordonnancements P-équitables, ordonnancement optimal, tolérance aux fautes, réseaux de Petri, ordonnancement multi modal.