THÈSE

pour l'obtention du Grade de
DOCTEUR DE L'UNIVERSITÉ DE POITIERS

Faculté des Sciences Fondamentales et Appliquées
et
École Nationale Supérieure de Mécanique et d'Aérotechnique
(Diplôme National - Arrêté du 25 avril 2002)

École Doctorale : Sciences Pour l'Ingénieur & Aéronautique

Secteur de Recherche : Sciences et technologies de l'information et de la communication
Informatique et applications

présentée et soutenue publiquement par :

Frédéric RIDOUARD

Équipe d'accueil : Équipe systèmes embarqués
Laboratoire d'accueil : **L**aboratoire d'**I**nformatique **S**cientifique et **I**ndustrielle (***LISI***)

Contributions à des problèmes d'ordonnancement en-ligne : l'ordonnancement temps réel de tâches à suspension et l'ordonnancement par une machine à traitement par lot

Directeur de Thèse : Pascal RICHARD

Soutenue le 14 novembre 2006 devant la commission d'examen

Président :
Patrick MARTINEAU Professeur à l'Université François-Rabelais de Tours
Rapporteurs :
Marie-Claude PORTMANN Professeur à l'École des Mines de Nancy
Yvon TRINQUET Professeur à l'Université de Nantes
Examinateurs :
Joël GOOSSENS Assistant Professor à l'Université Libre de Bruxelles
Francis COTTET Professeur d'Université à l'ENSMA
Pascal RICHARD Maître de conférences (HDR) à l'Université de Poitiers

Remerciements

Je tiens à remercier tout d'abord, Monsieur Guy PIERRA, directeur du Laboratoire d'Informatique Scientifique et Industrielle (LISI) de l'ENSMA, pour m'avoir accueilli au sein de son laboratoire et pour avoir mis à ma disposition, tous les moyens techniques et scientifiques nécessaires à mon travail ainsi qu'un environnement propice à mon développement personnel.

Je ne pourrai sans doute jamais assez remercier Monsieur Pascal RICHARD. Je le remercie d'avoir encadré mes travaux de DEA et aussi ceux de ma thèse. Toujours disponible, le pédagogue qu'il est, m'a toujours laissé une grande liberté dans mon travail. Nos discussions constructives ont été essentielles dans ma formation. Ses qualités aussi bien humaines que scientifiques ont été l'une des raisons principales de la qualité de notre collaboration ainsi que de mon épanouissement dans cette formation. Ce fut pour moi un immense honneur et un réel plaisir de profiter de ses conseils et de son soutien. Il est, pour moi, un parfait encadrant et quelqu'un qui restera toujours à part.

Je remercie
Monsieur Patrick MARTINEAU, Professeur à l'Université François-Rabelais de Tours
Madame Marie-Claude PORTMANN, Professeur à l'École des Mines de Nancy
Monsieur Yvon TRINQUET, Professeur à l'Université de Nantes
Monsieur Joël GOOSSENS, Assistant Professor à l'Université Libre de Bruxelles
Monsieur Francis COTTET, Professeur d'Université à l'ENSMA
pour m'avoir fait l'honneur de constituer mon jury de thèse ainsi que pour la qualité et la pertinence de leurs remarques constructives.

Merci aussi à tous les membres de l'équipe *systèmes embarqués*, Annie et Dominique GENIET, Emmanuel GROLLEAU et Karim TRAORE pour leurs remarques et conseils avisés sur mes travaux.

Je tiens à remercier également tous les membres de la famille LISI et en particulier, Claudine, Boulou, Dago, Fred, JCP, Mikky, Ricou, Tex, Yamine... pour tous les bons moments passés ensembles.

Je voudrais également souligner l'importance de Nelly, mais aussi de nos amis, Manuch, Marie, Mumu, JP, Loïc et Tibo, ainsi que de tous les membres de nos deux familles pour leur soutien moral indispensable qu'ils m'auront fourni tout au long de ces trois années. En particulier, merci à Maguy, pour avoir accepté de relire deux fois cette thèse.

Table des matières

Introduction

Chapitre 1

Introduction

1.1. Contexte

Le travail effectué pendant cette thèse s'est déroulé au sein du *Laboratoire d'Informatique Scientifique et Industrielle* (*LISI*). Ce laboratoire est composé en deux thématiques principales : *Ingénierie des données* et *systèmes embarqués (temps réel)*. Ce travail a été réalisé dans l'équipe systèmes embarqués, sous la direction de PASCAL RICHARD.

1.2. Présentation des sujets d'étude

Cette thèse est composée de deux sujets d'étude :

– l'ordonnancement temps réel des tâches à suspension ;

– l'ordonnancement en-ligne par une machine à traitement par lot.

L'ordonnancement temps réel est l'activité principale de l'équipe systèmes embarqués du *LISI*. Nous nous sommes intéressés à un problème d'ordonnancement particulier, celui des tâches à suspension.

Le second sujet d'étude, sujet d'ordonnancement classique, est intégré dans une activité contractuelle, le projet industriel W4L (*Workload For Labs*). L'étude d'une machine à traitement par lot est un problème théorique lié à ce projet.

1.3. Organisation du document

Ce document se décompose ainsi :

– le chapitre 2 présente la problématique et nos motivations qui ont conduit nos travaux ;

– la partie I développe notre étude sur l'ordonnancement temps réel de tâches à suspension ;

– la partie II traîte de l'ordonnancement classique par une machine à traitement par lot ;

– finalement, le chapitre 14 conclut notre travail et ouvre des perspectives de recherches.

Problématique

Chapitre 2

Problématique

La thématique de cette thèse se définit par l'étude de problèmes d'ordonnancement. Nous en avons étudié deux : l'ordonnancement temps réel de tâches à suspension et l'ordonnancement par une machine à traitement par lot. Ces deux problèmes sont basés sur la recherche et l'évaluation d'algorithmes en-ligne compétitifs. Pour analyser ces algorithmes nous utilisons, pour ces deux problèmes, la même technique : l'analyse de compétitivité, introduite par [SLE 85].

Nous présentons dans le paragraphe 2.1, la théorie de l'ordonnancement. Le paragraphe 2.2 traite des différences et similitudes notables entre l'ordonnancement classique et l'ordonnancement temps réel. Le paragraphe 2.3 définit et décrit les problèmes d'ordonnancement à une machine. Au paragraphe 2.4, les algorithmes en-ligne sont abordés. Finalement, nous présentons dans le paragraphe 2.5, notre technique d'analyse : l'analyse de compétitivité.

2.1. Théorie de l'ordonnancement

L'ordonnancement est présent et nécessaire dans tous les systèmes où des activités doivent être organisées, ordonnées et réparties entre plusieurs entités. L'ordonnancement est un domaine particulier de la recherche opérationnelle. La recherche opérationnelle (aussi appelée science du management ou science de la décision) regroupe l'ensemble des méthodes et techniques rationnelles d'analyse et de synthèse des phénomènes d'organisation utilisables pour élaborer de meilleures décisions. Cette problématique est multidisciplinaire puisqu'elle est traitée en mathématiques, en informatique et en sciences de gestion.

L'ordonnancement est utilisé dans plusieurs domaines qui sont aussi nombreux que variés (p. ex. l'usinage de pièces en production à la chaîne, la gestion des ressources humaines, l'exécution de programmes informatiques…). Pour ce qui est de la modélisation des systèmes, chaque activité quelle qu'elle soit, sera dénommée *tâche* et chaque entité, *ressource* ou *machine*. Ainsi, un problème d'ordonnancement revient à gérer l'exécution de tâches sur une (ou plusieurs) machine(s). Par exemple, pour l'usinage de pièces, les tâches sont les différentes opérations à effectuer sur la pièce (p. ex. soudage, perçage…) et les ressources représentent le personnel ainsi que les machines.

Ordonnancer, c'est planifier l'exécution d'un ensemble de tâches en leur allouant des ressources et en leur fixant des dates d'exécution. En effet, les tâches se concurrencent entre elles afin de pouvoir s'exécuter et ainsi de disposer prioritairement des ressources qui leur sont nécessaires. Par conséquent, un ordre ou des dates d'exécution doivent être attribués à chacune des tâches pour obtenir un ordonnancement. Pour une introduction plus complète sur l'ordonnancement, nous renvoyons à la lecture de [BAK 74, CAR 88, BLA 96, BRU 01, ESQ 99, PIN 01]. Dans le cadre de notre travail, nous nous sommes uniquement intéressés aux problèmes d'ordonnancement à une machine.

2.2. Ordonnancement classique et ordonnancement temps réel

Bien qu'utilisant des techniques de résolution similaire, voire les mêmes algorithmes *EDD* (*Earliest Due Date*) et EDF (*Earliest Deadline First*), l'ordonnancement classique et l'ordonnancement temps réel se distinguent par deux points :

– les problèmes fondamentaux de l'ordonnancement classique sont principalement des problèmes avec des tâches apériodiques (chaque tâche ne survient qu'une fois dans le système). Le problème central de l'ordonnancement temps réel considère des tâches qui reviennent périodiquement dans la vie du système. La périodicité de tâches change, en général, la nature des problèmes combinatoires [TOV 02] ;

– pour des raisons historiques : l'ordonnancement classique a été rapidement reconnu comme une théorie à part entière au sein de la communauté "recherche opérationnelle", alors que l'ordonnancement temps réel demeure un sous-problème dans la conception de système informatique temps réel. Bien sûr, l'implication de l'ordonnancement classique sur l'ordonnancement temps réel est indéniable [STA 95].

Notre position est que les problèmes d'ordonnancement temps réel doivent être simultanément perçus comme :

– une branche de l'ordonnancement classique, afin de profiter des approches et résultats connus en théorie de l'ordonnancement ;

– une branche de l'ingénierie des systèmes temps réel, pour ne pas se couper de la principale source d'application de l'ordonnancement temps réel, et donc de la principale source de nouveaux problèmes.

Remarque 1 *Les notations utilisées en ordonnancement classique et en ordonnancement temps réel ne sont malheureusement pas homogènes (p. ex. la durée d'exécution d'une tâche se note p_i en ordonnancement classique et C_i en ordonnancement temps réel, alors que C_i en ordonnancement classique définit la date de fin d'exécution d'une tâche qui est usuellement notée R_i en temps réel !). Nous avons essayé d'être généralement homogène dans ce mémoire, en privilégiant les notations de l'ordonnancement temps réel à chaque fois que possible, à l'exception des notations des problèmes de l'ordonnancement classique afin de ne pas dénaturer cette notation si pratique.*

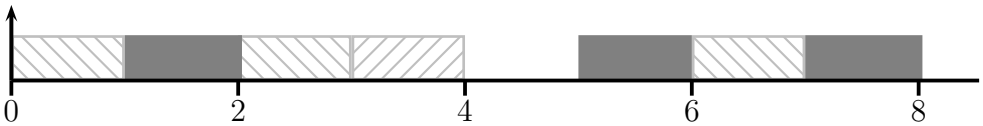

Figure 2.1. *Exemple d'un diagramme de* GANTT*, pour l'ordonnancement de tâches sur une machine.*

2.3. Problèmes d'ordonnancement

La théorie de l'ordonnancement comprend un nombre possiblement infini de problèmes, BAKER [BAK 74], BLAZEWICZ [BLA 96], COFFMAN [E.G 76], CONWAY et al. [CON 67], LENSTRA [LEN 77], PINEDO [PIN 01]... Ce paragraphe présente et définit les problèmes d'ordonnancement.

2.3.1. *Définition*

Soit M, une machine qui doit exécuter n tâches $\tau_i\,(i = 1, \ldots, n)$. Un ordonnancement se définit en allouant un ou plusieurs intervalles de temps sur la machine. L'ordonnancement peut être représenté par un diagramme temporel de GANTT (figure 2.1).

2.3.2. *Critères de performance*

Les critères de performance sont des critères que l'ordonnancement doit optimiser. Nous présentons dans ce paragraphe, une liste non exhaustive des critères de performance possibles d'un ordonnancement. A noter que suivant la nature du critère, l'optimisation est synonyme de minimisation ou de maximisation.

En ordonnancement classique, r_i note pour une tâche τ_i sa date d'activation, C_i sa date de fin d'exécution, $F_i = C_i - r_i$ son temps de réponse, $L_i = C_i - d_i$ son retard (décalage entre sa date de fin d'exécution et son échéance d_i) et U_i un booléen qui est à vrai (ou 1) si la tâche est en retard. Les critères de performance classiques sont :

- $C_{max} = \max_{1 \leq i \leq n} C_i$: la minimisation de la longueur d'ordonnancement ;
- $\sum_{1 \leq i \leq n} C_i$: la minimisation de la date moyenne de fin d'exécution ;
- $F_{max} = \max_{1 \leq i \leq n} F_i$: la minimisation du temps de réponse maximum ;
- $\sum_{1 \leq i \leq n} F_i$: la minimisation du temps de réponse moyen ;
- $L_{max} = \max_{1 \leq i \leq n} L_i$: la minimisation du retard maximum accumulé par les tâches ;
- $\sum_{1 \leq i \leq n} U_i$: la minimisation du nombre de tâches en retard ou de façon équivalente, $\sum_{1 \leq i \leq n}(1 - U_i)$: la maximisation du nombre de tâches n'étant pas en retard.

Il est également possible de pondérer le critère de performance, par un poids w_i associé à chaque tâche τ_i afin de tenir compte de l'importance des tâches.

2.3.3. *Notation des problèmes en ordonnancement classique*

La notation à 3-champs initiée par GRAHAM et al. [GRA 96, LAW 93] est utilisée pour décrire les problèmes d'ordonnancement. Cette notation se décompose en trois valeurs : $\alpha \mid \beta \mid \gamma$ et est celle principalement utilisée dans la littérature [BRU 01].

α **:** caractérise les machines sur lesquelles les algorithmes s'exécutent. Par exemple, pour un système avec une unique machine, $\alpha = 1$.

β **:** décrit les caractéristiques des tâches du système. Par exemple, si toutes les tâches ne sont pas à départ simultané $\beta = r_i$ (r_i signifiant que chaque tâche a sa propre date d'activation) et si de plus le système permet la préemption des tâches alors $\beta = r_i, pmtn$.

γ **:** précise la nature du critère de performance étudié. Par exemple pour la maximisation du nombre de tâches qui ne sont pas en retard, $\gamma = \sum_{1 \leq i \leq n}(1 - U_i)$

Remarque 2 *Pour le paramètre β, nous avons introduit la notion de préemption (ou non) des tâches. Il existe plusieurs possibilités :*

– Non préemptif : les tâches ne peuvent être interrompues lors de leur exécution qu'à des moments spécifiques et que sur l'initiative de la tâche elle-même ;

– Préemptif-Continue : les tâches peuvent être interrompues à n'importe quel instant et le processeur affecté à une autre tâche. Quand la tâche reprendra le cours de son exécution, elle le reprendra là où elle l'avait laissé. Cet ordonnancement se note "pmtn" dans le champ β ;

– Préemptif-redémarre (ou recommence) : les tâches peuvent être interrompues à n'importe quel instant et le processeur affecté à une autre tâche. Quand la tâche reprendra le cours de son exécution, elle le reprendra au début de l'exécution comme si elle ne l'avait jamais démarré. Cet ordonnancement se note "rstart" dans le champ β.

2.4. Algorithmes hors-ligne/en-ligne

Les algorithmes d'ordonnancement peuvent être classés dans deux catégories, les algorithmes d'ordonnancement hors-ligne et en-ligne.

Les *algorithmes d'ordonnancement hors-ligne* construisent la séquence d'ordonnancement complète sur les bases des paramètres temporels de l'ensemble des tâches de la configuration. L'ordonnancement est construit complètement avant l'exécution de l'application.

Les *algorithmes d'ordonnancement en-ligne* choisissent, quant à eux, dynamiquement la prochaine requête à exécuter en fonction des paramètres des tâches prêtes à l'exécution. Ils ne connaissent à un instant t que les tâches dont les dates d'activation sont antérieures à t.

Il existe, [PRU 04, SGA 98], toutefois plusieurs paradigmes pour caractériser les problèmes d'ordonnancement en-ligne :

– **Ordonnancement une à une des tâches (ou ordonnancement de liste).** Dès qu'une tâche arrive, ses caractéristiques sont connues. Ensuite les tâches sont ordonnées dans une liste puis présentées une à une à l'ordonnanceur. La seule tâche qui peut être assignée à une machine est la première tâche de la liste. Une tâche peut être retardée mais aucune machine ne restera inoccupée s'il reste des tâches à ordonnancer dans la liste. L'arrivée d'une nouvelle tâche dans la liste ne peut pas stopper l'assignement d'une tâche à une machine.

– **Temps processeurs inconnus.** Le temps processeur nécessaire à une tâche n'est connu que lorsqu'elle a fini son exécution mais ses autres caractéristiques sont connues dès son apparition dans le système. Toutes les tâches qui sont prêtes à être exécutées sont disponibles pour l'ordonnanceur, elles peuvent être exécutées ou retardées. Si le système le permet, l'algorithme peut décider de préempter les tâches à n'importe quel instant.

– **Les tâches arrivent dans le temps.** Ce paradigme est quasiment similaire au précédent. Mais là, le temps processeur et la date d'arrivée ne sont connus qu'à l'arrivée de la tâche.

– **Intervalle d'ordonnancement.** Se dit d'un ordonnancement dont les tâches n'ont qu'un intervalle donné pour s'exécuter. Si l'ordonnanceur voit qu'une tâche ne finira pas dans les délais qui lui sont imposés, elle ou une autre peut être rejetée.

Les algorithmes d'ordonnancement peuvent disposer d'autres caractéristiques :

– Un algorithme d'ordonnancement en-ligne est *conservatif*, s'il ne laisse jamais le processeur inoccupé lorsque des tâches sont prêtes à s'exécuter.

– Un algorithme *clairvoyant* est un algorithme qui connaît toutes les caractéristiques des tâches à venir.

– Un algorithme *déterministe* est un algorithme qui ne fait intervenir aucune composante aléatoire dans la prise de décision d'ordonnancement. Ainsi une configuration ordonnancée plusieurs fois par un même algorithme déterministe donnera toujours le même ordonnancement et donc la même planification de tâches. Ce type d'algorithme est l'opposé des algorithmes *randomisés* qui eux prennent leurs décisions dans le temps avec un choix aléatoire.

Définition 1 *Un algorithme A est optimal pour un problème d'ordonnancement si pour chaque configuration de tâches, aucun autre algorithme ne peut avoir une meilleure performance que celle obtenue par A.*

Remarque 3 *Il est à noter que pour un problème d'ordonnancement étudié, plusieurs algorithmes peuvent être optimaux.*

2.5. Ordonnancement en-ligne : Simulation/Analyse de compétitivité

2.5.1. *Introduction*

La validation des algorithmes en-ligne est une étape importante. Elle permet, pour un problème donné, de trouver quelle est la meilleure stratégie d'ordonnancement en-ligne ainsi que de quantifier la qualité de cette méthode en la comparant à la solution optimale. Deux types de validation peuvent être appliqués :

– Comparer et évaluer les différentes stratégies des méthodes en-ligne entre elles. Nous exécutons sur une même configuration de tâches, les différents algorithmes puis nous étudions les résultats obtenus. Ce type de comparaison est appelé la *simulation*.

– Evaluer le comportement d'un algorithme en-ligne en son pire cas d'exécution en le comparant à un algorithme hors-ligne optimal. Cette méthode est l'*analyse de compétitivité* (*Competitive Analysis*) et définit la garantie de performance de l'algorithme en-ligne quelle que soit l'instance du problème qui sera résolue.

2.5.2. *La simulation*

La simulation permet de comparer et d'évaluer des algorithmes en-ligne entre eux. Elle consiste à définir un modèle stochastique muni d'une loi de probabilité. Avec cette loi de distribution aléatoire, des configurations de tâches sont générées, puis elles sont ordonnancées par les algorithmes étudiés. En dernier lieu, les réactions et les résultats des ordonnancements de chacun des algorithmes sont relevés. Cette opération est répétée suffisamment de fois pour obtenir une étude statistique pertinente et ainsi des comparatifs entre algorithmes satisfaisants.

Mais cette méthode a ses limites qui pénalisent l'exploitation de ses résultats. En effet, les données obtenues par simulation sont forcément dépendantes du modèle stochastique utilisé [KAR 92]. De plus, cette méthode est adaptée si et seulement si les lois de distribution (qui sont estimées généralement sur la base d'observations passées) modélisent toujours l'arrivée des nouvelles tâches. Ce qui n'est pas toujours possible voire réalisable avec des problèmes d'ordonnancement en-ligne.

2.5.3. *L'analyse de compétitivité*

L'*analyse de compétitivité* [BOR 98] est une méthode dont les premiers résultats datent de 1985. Ils sont l'œuvre de SLEATOR et TARJAN [SLE 85]. Cette méthode compare le comportement d'un algorithme en-ligne à celui d'un algorithme clairvoyant optimal (dit l'adversaire). Cette comparaison s'effectue sur les configurations de tâches où l'algorithme en-ligne atteint sa pire performance vis-à-vis du critère de performance étudié. L'adversaire est celui qui génère les configurations de tâches sur lesquelles les algorithmes sont étudiés. Par conséquent, un bon adversaire est celui qui générera les configurations de tâches sur lesquelles l'algorithme en-ligne atteindra ses pires performances (par rapport au critère choisi), tandis que l'adversaire ordonnancera de façon optimale les configurations nouvellement générées.

Il existe deux types d'adversaires mais qui sont équivalents du point de vue de l'analyse de compétitivité :

L'inconscient (*oblivious*) : il génère, en avance, une configuration de tâches, en s'assurant que l'algorithme en-ligne atteint sa pire performance sur cette configuration alors que lui la sert de façon optimale.

L'évolutif (*on-line adaptative*) : il génère quelques tâches et, suivant les réactions de l'algorithme en-ligne, complète (ou pas) cette configuration avec l'arrivée de nouvelles tâches. La configuration générée reste ordonnancée de façon optimale par l'adversaire, alors que l'algorithme en-ligne atteint sa pire performance.

Nous présentons l'analyse de compétitivité d'un point de vue analytique :

Définition 2 *Soit A un algorithme en-ligne, la performance obtenue par l'algorithme A en ordonnançant une configuration I pour un critère de performance σ, se note $\sigma_A(I)$.*

Définition 3 *La performance obtenue par l'algorithme clairvoyant et optimal hors-ligne en ordonnançant une configuration I, pour un critère σ, se note $\sigma^*(I)$.*

Définition 4 *Un algorithme en-ligne A est dit c-compétitif pour un critère donné σ, si c est une constante positive et si pour toute configuration de tâches I, l'inégalité suivante est respectée :*

$$\sigma_A(I) \leq c\,\sigma^*(I)$$

Deux remarques complémentaires sont à apporter à la définition 4 :

Remarque 4 *Aucun algorithme ne peut faire mieux que l'algorithme optimal. Ainsi sa performance est toujours égale ou meilleure que celle obtenue par l'algorithme en-ligne.*

Remarque 5 *La définition 4 montre que la performance atteinte par l'algorithme A est toujours au pire c fois supérieure que celle obtenue par l'adversaire par rapport au critère étudié. C'est-à-dire que pour chaque configuration, la performance de l'algorithme en-ligne varie mais ne dépasse pas c fois la valeur obtenue par l'adversaire.*

Pour finir la présentation de l'analyse de compétitivité, deux catégories de problèmes d'ordonnancement doivent être dissociées suivant la nature du critère à optimiser : les problèmes de minimisation et ceux de maximisation. Cette différenciation intervient pour l'établissement du ratio de compétitivité (*competitive ratio*). Ce ratio permet de caractériser les algorithmes en-ligne, de quantifier leur qualité (ou médiocrité). En effet, l'analyse de compétitivité ne se contente pas de donner une réponse binaire : l'algorithme est optimal ou non compétitif. Elle quantifie la qualité (ou médiocrité) de l'algorithme. Et c'est le ratio de compétitivité qui transcrit cette quantification. Il se détermine dans le pire cas d'exécution de l'algorithme en-ligne, c'est l'unique valeur d'analyse. Le ratio de compétitivité d'un algorithme en-ligne A se note c_A et définit la valeur c présente dans la définition 4.

2.5.4. *Le problème de minimisation*

Lorsqu'un problème d'ordonnancement est caractérisé par un critère à minimiser, le ratio de compétitivité d'un algorithme en-ligne est présenté par la définition 5.

Définition 5 *Si A est un algorithme en-ligne qui minimise un critère de performance σ alors, le ratio de compétitivité c_A de A qui représente son pire cas d'exécution vaut :*

$$c_A = \sup_{toutI} \frac{\sigma_A(I)}{\sigma^*(I)}$$

Où I désigne les configurations de tâches.

Remarque 6 *Puisque le critère d'optimisation est un critère à minimiser, la performance obtenue par l'algorithme optimal sera toujours inférieure ou égale à celle obtenue par l'algorithme en-ligne. Ainsi le pire cas d'exécution de l'algorithme correspond à la configuration de tâches I, pour laquelle le ratio $\frac{\sigma_A(I)}{\sigma^*(I)}$ est le plus important, d'où l'utilisation de la borne supérieure.*

Comme la performance de l'algorithme en-ligne est toujours supérieure ou égale à celle obtenue avec l'algorithme optimal, le ratio de compétitivité d'un algorithme en-ligne est toujours supérieur à 1. Et les deux cas limites sont les suivants :

– Si $c_A = 1$ alors l'algorithme A est dit optimal pour le critère étudié.

– Si $c_A \neq c$ pour toute constante c (ce qui est équivalent à c_A est égal à une fonction en n, où n est le nombre de tâches de la configuration testée, alors l'algorithme A est dit non compétitif pour le critère étudié.

L'étude d'un problème d'ordonnancement avec un critère de minimisation se fait par deux actions :

– Soit l'étude consiste à établir une borne inférieure du ratio de compétitivité valable pour tous les algorithmes en-ligne. Dans ce cas, l'ensemble de tous les algorithmes en-ligne de la classe étudiée (p. ex. l'ensemble des algorithmes en-ligne, conservatif et préemptif) est considéré. Il faut alors chercher une configuration telle que tous les algorithmes en-ligne de la classe soient menés à leur pire cas d'exécution sur cette configuration.

– Soit l'étude consiste à analyser un algorithme en-ligne de notre choix. Ainsi, l'adversaire génère des configurations de tâches pour amener l'algorithme en-ligne à son pire cas d'exécution. Cette méthode cherche à construire une borne supérieure du ratio de compétitivité de l'algorithme étudié.

Remarque 7 *Si nous établissons un algorithme en-ligne tel que la borne inférieure est égale à la borne supérieure alors l'algorithme fait partie des meilleurs algorithmes en-ligne pour ce problème.*

2.5.5. *Le problème de maximisation*

Définition 6 *Si A est un algorithme qui maximise un critère de performance alors, le ratio de compétitivité c_A de A qui représente son pire cas d'exécution vaut :*

$$c_A = \inf_{toutI} \frac{\sigma_A(I)}{\sigma^*(I)}$$

Où I désigne les configurations de tâches.

Étant donné que le critère à optimiser est un critère de maximisation, la performance obtenue par l'algorithme optimal est toujours supérieure ou égale à celle obtenue par l'algorithme en-ligne. Ainsi, le ratio de compétitivité est inférieur ou égal à 1 :

– Si $c_A = 1$ alors l'algorithme A est dit optimal pour le critère étudié.

– Si $c_A = 0$ alors l'algorithme A est dit non compétitif pour le critère étudié.

L'étude d'un problème d'ordonnancement avec un critère de maximisation est différente de celle menée pour le problème de minimisation :

– Soit l'étude consiste à établir une borne inférieure du ratio de compétitivité pour *un* algorithme en-ligne. Il faut chercher des configurations de tâches pour que le ratio de compétitivité de l'algorithme en-ligne augmente.

– Soit l'étude consiste à établir une borne supérieure pour une classe d'algorithmes. Il faut chercher des configurations de tâches telles que sur ces configurations, le ratio de compétitivité de tous les algorithmes est inférieur à une même valeur.

Remarque 8 *De même que, concernant le problème de minimisation, si pour un algorithme en-ligne tel que la borne inférieure est égale à la borne supérieure alors il fait partie des meilleurs algorithmes en-ligne pour ce problème.*

2.5.6. *Principaux résultats*

Le tableau 2.1 (présenté dans [RID 03]) synthétise l'ensemble des résultats connus pour les problèmes d'ordonnancement à une machine. Lorsque la borne inférieure et la borne supérieure sont égales alors un algorithme avec les meilleures garanties possibles de performance est connu. Lorsque pour un problème, les deux bornes ne sont pas égales, le problème est alors partiellement ouvert. De plus, si les bornes sont égales à 1, alors l'algorithme est optimal. Finalement, s'il n'y a pas de résultat dans certaines d'entre elles, c'est que le problème est ouvert.

2.5.7. *Conclusion*

L'inconvénient de l'analyse de compétitivité est de ne capturer la performance d'un algorithme en-ligne que par son ratio de compétitivité. Très souvent, cela conduit à une vision

Problèmes	Modèles	Bornes Inférieures		Bornes Supérieures	
$\min \sum C_j$	Préemption	1		1	
	Standard	2	[PHI 95]	2	[PHI 95, HOO 96]
	Redémarrage	1.2108	[EPS 03]	3/2	[VAN 02]
$\min \sum F_j$	Préemption	1		1	
	Standard	$\Omega(n)$			
	Redémarrage	$\Omega(\sqrt{n})$	[EPS 03]	$\Theta(n)$	[EPS 03]
$\min \sum w_j C_j$	Préemption	1.0730	[EPS 03]	2	
	Standard	2	[PHI 95]	2	[AND 02]
	Redémarrage	1.2232	[EPS 03]	2	[AND 02]
$\min \sum w_j F_j$	Préemption	2	[EPS 03]	$\Theta(n)$	[EPS 03]
	Standard	$\Omega(n)$			
	Redémarrage	$\Omega(n)$	[EPS 03]	$\Theta(n^2)$	[EPS 03]
$\min L_{max}$ ($d_i \leq 0$)	Préemption	1		1	
	Standard	1.618	[HOO 00]	1.618	[HOO 00]
	Redémarrage	3/2	[AKK 00]	3/2	[AKK 00]
$\max \sum(1 - U_j)$	Préemption			0	[BAR 94, BAR 01]
	Standard			0	[BAR 94, BAR 01]
	Redémarrage	1/2	[HOO 00]	1/2	[HOO 00]

Tableau 2.1. *Garanties de performance des algorithmes d'ordonnancement en-ligne à une machine [RID 03].*

très pessimiste du comportement de l'algorithme évalué. Plusieurs relaxations de l'analyse de compétitivité ont été proposées dans la littérature afin de limiter cette vision pessimiste comme l'augmentation de la vitesse des machines utilisées par l'algorithme en-ligne par rapport à celles utilisées par l'adversaire claivoyant [KAL 95]. Cette technique est appelée *technique d'augmentation de ressource* et conduit à des résultats très intéressants tant en ordonnancement classique qu'en ordonnancement temps réel.

2.6. Bibliographie

[AKK 00] VAN DEN AKKER M., HOOGEVEEN H., VAKHANIA N., « Restarts can help in the on-line minimization of the maximum delivery time on a single machine », *proc. European Symposium on Algorithms*, p. 427-436, 2000.

[AND 02] ANDERSON E., POTTS C., « On-line scheduling of a single machine to minimize total weighted completion time », *in : proc. ACM-SIAM Symposium on Discrete Algorithms*, p. 548-557, 2002.

[BAK 74] BAKER K., *Introduction to sequencing and scheduling*, John Wiley & Sons, New-York, 1974.

[BAR 94] BARUAH S., HARITSA J., SHARMA N., « On-line scheduling to maximise task completions », *in : proc. Real-Time Systems Symposium*, p. 228-236, 1994.

[BAR 01] BARUAH S., HARITSA J., SHARMA N., « On-line scheduling to maximise task completions », *The Journal of Combinatorial Mathematics and Combinatorial Computing*, vol. 39, p. 65-78, 2001.

[BLA 96] BLAZEWICZ J., ECKER K., PESCH E., SCHMIDT G., WEGLARZ J., *Scheduling in Computer and Manufacturing Systems*, Springer Verlag, Berlin, 1996.

[BOR 98] BORODIN A., R.EL-YANIV, *Online Computation and Competitive analysis*, Cambridge University Press, 1998.

[BRU 01] BRUCKER P., *Scheduling Algorithms,* (Third edition) Springer Verlag, 2001.

[CAR 88] CARLIER J., CHRÉTIENNE P., *Problèmes d'ordonnancement, modélisation, complexité, algorithmes,* Masson, 1988.

[CON 67] CONWAY R., MAXWELL W., MILLER L., *Theory of Scheduling,* Addison Wesley, Reading, Mass., USA, 1967.

[E.G 76] E.G. COFFMAN J., *Scheduling in Computer and Job Shop Systems,* Wiley, New York, 1976.

[EPS 03] EPSTEIN L., VAN-STEE R., « Lower bounds for on-line single-machine scheduling », *Theoretical Computer Science,* , n°299, p. 439-450, 2003.

[ESQ 99] ESQUIROL P., LOPEZ P., *L'ordonnancement,* Economica, Paris, 1999.

[GRA 96] GRAHAM R., LAWLER E., LENSTRA J., KAN A. R., « Optimisation and approximation in deterministic sequencing and scheduling : A survey », *Annals of Discrete Mathematics*, vol. 5, p. 287-326, 1996.

[HOO 96] HOOGEVEEN H., VESTJENS A., « Optimal on-line algorithms for single-machine scheduling », *in : proc. 5th Conference on Integer Programming and Combinatorial Optimization*, p. 404-414, 1996.

[HOO 00] HOOGEVEEN H., POTTS C. N., WOEGINGER G. J., « On-line scheduling on a single machine : maximizing the number of early jobs », *Operations Research Letters*, vol. 27, p. 193-197, 2000.

[KAL 95] KALYANASUNDARAM B., PRUHS K., « Speed is as powerful as clairvoyance », *proc. IEEE Foundations of Computer Science*, p. 214-223, 1995.

[KAR 92] KARP R., « On-line algorithms versus off-line algorithms : How much is it worth to know the future ? », *Algorithms, Software, Architecture, IFIP Transactions A-12,Vol. Information Processing*, vol. 1, p. 416-429, 1992.

[LAW 93] LAWLER E., LENSTRA J., KAN A. R., SHMOYS D., « Sequencing and scheduling : algorithms and complexity », *in : Handbooks in Operations Research and Management Science. North Holland*, vol. 4, p. 445-552, 1993.

[LEN 77] LENSTRA J., « Sequencing by enumerative methods », *Mathematical Centre Tracts*, vol. 69, page , 1977.

[PHI 95] PHILLIPS C., STEIN C., WEIN J., « Scheduling jobs that arrive over time », *in : proc. 4th Workshop on Algorithms and Data Structures, LNCS 955, Springer Verlag*, p. 86-97, 1995.

[PIN 01] PINEDO M., *Scheduling : Theory, Algorithms, and Systems (2nd Edition),* Prentice Hall, Englewood Cliffs, 2001.

[PRU 04] PRUH K., SGALL J., TORNG E., « *Handbook of Scheduling : Algorithms, Models, and Performance Analysis* », vol. 2, Chapitre Online scheduling, Editor : Joseph Y-T. Leung, CRC Press, Boca Raton, FL, USA, 2004.

[RID 03] RIDOUARD F., RICHARD P., COTTET F., « Algorithmes d'ordonnancement en-ligne à une machine », *École d'Automne de Recherche Opérationnelle, Tours*, vol. 1, n°28–31, octobre 2003.

[SGA 98] SGALL J., « On-line scheduling - a survey », *In A. Fiat and G. Woeginger, editors, On-Line Algorithms : The State of the Art, Lecture Notes in Computer Science, Springer Verlag*, vol. 1442, p. 196-231, 1998.

[SLE 85] SLEATOR D. D., TARJAN R. E., « Amortized efficiency of list update and paging rules », *Communication of the ACM 28*, vol. 2, p. 202-208, 1985.

[STA 95] STANKOVIC J., SPURI M., DINATALE M., BUTTAZZO G., « Implications of classical scheduling results for real-time systems », *IEEE Computer*, vol. 28, n°6, p. 16-25, 1995.

[TOV 02] TOVEY A., « Tutorial on computational complexity », *Interface*, vol. 32, n°3, p. 30-61, Mai-Juin 2002.

[VAN 02] VAN-STEE R., POUTRÉ J. L., « Minimizing total completion time on-line on a single machine, using restarts », *10th European Symposium on Algorithms, Lecture Notes in Computer Science, Springer Verlag*, 2002.

Première partie :

Ordonnancement des tâches à suspension

PREMIÈRE PARTIE

Ordonnancement des tâches à suspension

Cette partie présente l'ensemble de nos travaux sur l'ordonnancement temps réel de tâches à suspension :

– Le chapitre 4 expose les principaux concepts de l'ordonnancement temps réel.

– Le chapitre 5 est un état de l'art de notre problème d'ordonnancement.

– Dans le chapitre 6, nous présentons nos résultats sur les difficultés de l'ordonnancement de tâches à suspension : la complexité du problème…

– Le chapitre suivant, chapitre 7, développe notre étude sur la compétitivité des algorithmes en-ligne pour l'ordonnancement des tâches à suspension.

– Le chapitre 8 conclut notre recherche sur les tâches à suspension par l'analyse des tests d'ordonnançabilité existants pour ce problème d'ordonnancement.

– En conclusion, au chapitre 9, nous résumons l'ensemble de nos résultats et proposons plusieurs axes pour de futures recherches.

Table des matières

Chapitre 3

Introduction sur les tâches à suspension

3.1. Les systèmes temps réel

Les systèmes temps réel sont des systèmes informatiques qui doivent réagir à des évènements tout en respectant des contraintes temporelles. Trois catégories de systèmes informatiques sont généralement distinguées dans la littérature :

– les systèmes transformationnels : ils sont composés entre autres des activités de calculs et de gestion des bases de données. Ces systèmes récupèrent à l'initialisation l'ensemble des données nécessaires. Leurs temps de calculs sont non contraints.

– les systèmes interactifs : un système est dit interactif s'il implique l'intervention de l'utilisateur dans la vie du système, au-delà de l'initialisation. Ces systèmes sont composés entre autres des logiciels bureautiques. Le temps n'intervient pas en tant que contrainte mais dans un aspect de confort de travail ou qualité de service.

– les systèmes réactifs [HAR 85] : un système réactif, comme présenté par la figure 3.1, est un système répondant constamment aux sollicitations de son environnement qu'il contrôle en produisant des actions sur celui-ci. Les systèmes temps réel appartiennent à cette catégorie.

De nombreuses définitions différentes existent pour les systèmes temps réel dans la littérature ([CNR 88], [DOR 91], [LAP 91]…). Nous utilisons la définition employée par J.A. STANKOVIC : *un système temps réel est un système pour lequel sa correction ne dépend pas seulement des résultats, mais également du temps auquel ils sont fournis* [STA 88]. Ainsi, les systèmes temps réel sont des systèmes réactifs [HAR 85] pour lesquels des contraintes temporelles doivent être respectées. Si les contraintes temporelles de l'application ne sont pas respectées, le système subit alors une *défaillance*, on parle aussi de *faute temporelle*. Deux types de contraintes temporelles existent :

– contraintes temporelles souples (ou relatives) : pour les systèmes *souples*, le non-respect d'une contrainte mène à une dégradation progressive du fonctionnement du système. Ce type de défaillance est tolérable ;

– contraintes temporelles impératives : pour les systèmes *durs* (ou *critiques*), le non-respect d'une contrainte temporelle n'est pas envisageable, il remet en cause l'intégrité même

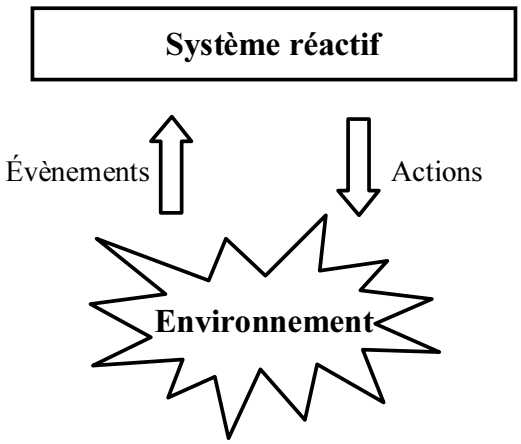

Figure 3.1. *Système réactif*

du système.

Cette définition du temps réel introduit la notion de *temps*. Cette notion du temps signifie que le calcul effectué par le système ne peut pas être considéré comme correct ; s'il n'est pas produit au bon instant, le système le considère comme faux.

La notion de temps est très importante car elle permet d'exprimer les caractéristiques et les contraintes du système. Le temps peut être représenté de deux façons différentes [KOP 83] :

– Le temps continu : est un temps que l'on suppose généralement dense (et complet), au sens mathématique du terme (plus précisément, le temps prend toutes les valeurs réelles positives). Ainsi, il est toujours possible de déterminer entre deux instants t et t_1, un troisième instant t_2. Il n'est par conséquent pas possible d'observer la simultanéité d'événements ;

– Le temps discret (ou logique) [BAR 90] : correspond à une succession d'instants discrets, prédéfinis et ordonnés. Ce temps n'est pas dense, il n'est ainsi pas possible de définir un instant entre deux instants consécutifs. Un instant est représenté mathématiquement par un entier naturel et ainsi le temps par l'ensemble des entiers naturels. La simultanéité des événements est donc possible.

Le temps continu est utilisé pour représenter les caractéristiques de l'environnement et du procédé à l'intérieur de l'environnement. Le temps discret est utilisé par le système informatique, pour représenter les caractéristiques des différentes actions à effectuer ainsi que les délais à tenir.

En ce qui concerne les systèmes temps réel durs, il faut s'assurer que toutes les contraintes temporelles des tâches sont respectées et ce quel que soit le cas de figure. Il existe plusieurs techniques de validation d'une configuration de tâches sur un système. Notamment, celle qui consiste à établir un test d'ordonnançabilité [BAR 04]. Pour établir un test d'ordonnançabilité, il faut caractériser le pire comportement du système, puis établir un algorithme que le système doit respecter (ou chacune des tâches du système doit respecter) pour que le système soit dit valide (c.-à-d. que toutes les contraintes temporelles des tâches soient respectées).

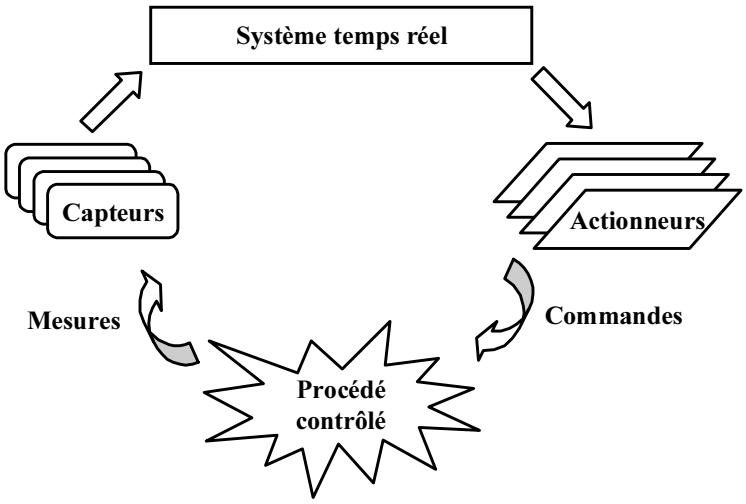

Figure 3.2. *Système temps réel*

Les systèmes temps réel doivent avoir certaines caractéristiques [COT 05] :

efficacité : un système temps réel doit être efficace (avec une complexité moindre) ;

fiabilité : un système temps réel doit s'assurer que toutes les contraintes temporelles sont respectées ;

prédictibilité : un système temps réel, s'il se retrouve dans un état déjà rencontré, devra toujours prendre la même décision.

Un système temps réel doit s'assurer que le procédé contrôlé qu'il gère, reste continuellement dans un fonctionnement dit *normal* (c.-à-d. qu'il reste en adéquation avec les lois physiques qui régissent son comportement dans son environnement). Dans les systèmes temps réel, comme le montre la figure 3.2, le système temps réel récupère par l'intermédiaire d'un ensemble de capteurs présents sur le procédé des informations relatives à l'état du système. Le système temps réel analyse les résultats et interagit, si nécessaire, par un ensemble d'actionneurs sur le procédé contrôlé. Dans ces systèmes, c'est le procédé contrôlé qui impose sa vitesse de fonctionnement (on parle aussi de *dynamique du procédé*) pour conserver son intégrité et son bon fonctionnement dans l'environnement. Cette caractéristique temporelle peut varier de la milliseconde (p. ex. les avions de chasse) à l'heure ou plus (p. ex. contrôle de réactions chimiques).

Les systèmes temps réel prennent une place de plus en plus importante dans notre société car les applications nécessitant un système informatique sont souvent contrôlées par tout ou partie par un système temps réel. Ainsi, de tels systèmes servent à contrôler des centrales nucléaires, des sites complexes de production, des applications militaires, des missions spatiales, mais aussi dans la vie de tous les jours, les voitures, les téléphones portables, les avions… Un exemple complet d'une application temps-réel en automobile est détaillé dans [SIM 05].

Une application est souvent naïvement qualifiée de temps réel dès lors qu'elle se montre « rapide ». Mais cette assertion est fausse pour les systèmes dont les délais s'expriment en heures (tels que les systèmes contrôlant certaines réactions chimiques) et qui sont qualifiés de système temps réel.

Comme pour tout système informatique, l'architecture matérielle d'un système temps réel est de différentes formes : centralisé ou distribué, monoprocesseur ou multiprocesseur...

3.2. Architecture logicielle des systèmes temps réel

Il existe trois différentes architectures logicielles des systèmes temps réel :

– un *noyau temps réel* : c'est l'architecture logicielle minimum pour un système temps réel. Cette architecture gère uniquement les tâches et les communications inter-tâches ;

– un *exécutif temps réel* : c'est une surcouche du noyau temps réel qui comprend des primitives/modules pour faciliter la conception de l'application temps réel ; il inclut des implémentations de fonctions additionnelles comme la gestion des entrées/sorties, la gestion avancée des tâches, la gestion de systèmes de fichiers, la communication de messages, ou la gestion des ressources. Pour le développement d'une application, on a besoin d'une part, d'une machine hôte et de son environnement de développement pour sa conception, ainsi que d'un système cible, sur lequel l'application temps réel et l'exécutif seront téléchargés pour son exécution.

– un *système d'exploitation temps réel* est un exécutif pour le cas particulier où le système hôte et le système cible ne font qu'un. On a donc ici un environnement de développement natif.

L'architecture logicielle que nous étudions est celle d'un exécutif temps réel. Un exemple d'exécutif temps réel est OSEK/VDX[1], présenté dans [TRI 05]. La figure 3.3 illustre la structure d'un exécutif temps réel. Ainsi, l'exécutif est appelé soit lors d'un événement, soit en fonction du temps, soit par des tâches nécessitant un service de l'exécutif. L'ordonnanceur gère l'occupation du ou des processeurs. Il utilise un algorithme d'ordonnancement qui gère l'allocation des tâches sur le processeur. Les agences gèrent les composants de base de l'exécutif : tâches, ressources...

Le comportement concurrent des évènements externes et des actions d'un système temps réel, fait de lui un système fortement parallèle. L'architecture logicielle la mieux adaptée pour respecter ce comportement parallélisé est une architecture multitâche. Ainsi, chaque action du système (acquisition de mesures, calculs...) est supportée par une tâche. La tâche est l'entité de base des systèmes temps réel, elle représente le travail que doit effectuer le système pour effectuer l'action à laquelle elle est associée. Chaque tâche a ses contraintes temporelles imposées qu'elle doit respecter (p. ex. un délai pendant lequel elle peut s'exécuter et qu'elle ne peut pas dépasser).

1. OSEK/VDX : Offene Systeme und deren Schnittstellen für dir Elektronik im Kraftfahrzeug / Vehicle Distributed eXecutive

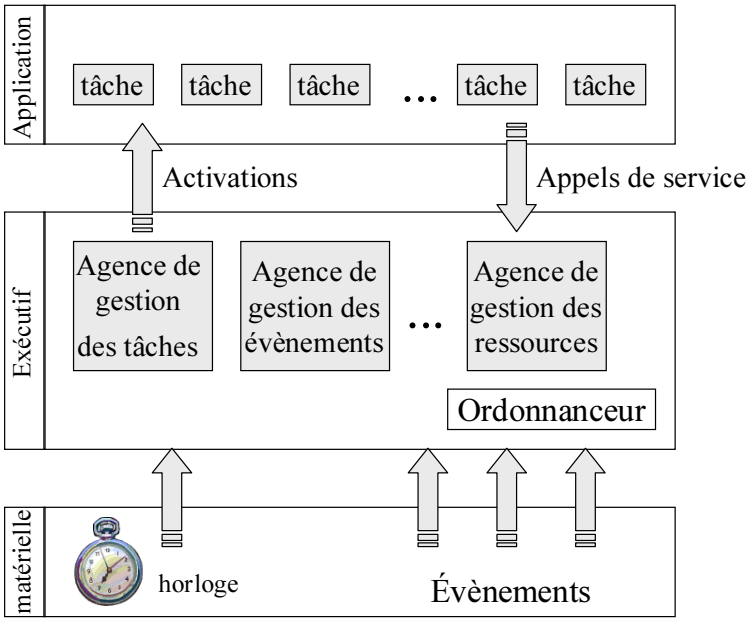

Figure 3.3. *Exécutif temps réel*

Les systèmes temps réel sont ainsi formés d'un ensemble de tâches (appelé aussi *configuration*) qui s'exécutent sur un exécutif temps réel. L'ensemble de tâches caractérise toutes les opérations à mener pour piloter le procédé contrôlé et l'exécutif temps réel est celui qui les ordonnance en leur affectant des priorités. À chaque instant la tâche la plus prioritaire est exécutée. La particularité des tâches temps réel réside dans le fait qu'elles sont récurrentes. C'est-à-dire que chaque tâche se réveille périodiquement (ou sporadiquement) dans le système pour être exécutée.

3.3. Ordonnancement temps réel

Afin de respecter les contraintes temporelles des tâches, le concepteur d'une application temps réel doit définir une méthode d'ordonnancement.

Définition 7 *une méthode d'ordonnancement consiste en :*

– un algorithme d'ordonnancement qui sera exécuté en-ligne ;

– un test d'ordonnançabilité dont l'objet est de vérifier que les tâches respectent leurs contraintes temporelles.

Dans de nombreux cas, l'ordonnancement est dit dirigé par les priorités. C'est-à-dire que les priorités sont attribuées aux tâches via un algorithme dit d'ordonnancement. L'ordonnanceur de l'exécutif utilise ces priorités pour construire la séquence d'exécution de l'application. Le *test d'ordonnançabilité* est un algorithme qui donne une condition suffisante ou dans le meilleur cas nécessaire et suffisante d'ordonnançabilité pour une configuration

de tâches par l'algorithme d'ordonnancement choisi précédemment. C'est-à-dire qu'un test d'ordonnançabilité permet de vérifier en amont à un ordonnancement que toutes les contraintes temporelles sont respectées par le choix de priorités qui est fait par l'algorithme d'ordonnancement.

La périodicité des tâches est l'unique spécificité de l'ordonnancement temps réel par rapport à l'ordonnancement classique. Cette différence peut paraître mineure mais elle change la nature combinatoire des problèmes d'ordonnancement. Mais l'ordonnancement temps réel repose sur les mêmes techniques algorithmiques de résolution de problèmes combinatoires [STA 95].

3.4. Le problème étudié : ordonnancement de tâches à suspension

La problème étudié par la suite, est celui de l'ordonnancement en-ligne de tâches à suspension.

Ainsi, Nous considérons, dans la suite de cette partie, les tâches pouvant se suspendre durant leur exécution afin de réaliser des opérations d'entrée/sortie ou bien des calculs sur un processeur spécialisé. Comme le présente la figure 3.4, les tâches sont exécutées sur le processeur principal, mais les tâches à suspension font appel pendant leur exécution à un processeur spécialisé pour effectuer différentes opérations. Durant sa suspension, la tâche n'utilise plus le processeur. En effet, elle attend la fin de l'exécution de cette opération pour terminer sa propre exécution. Ainsi, comme elle n'utilise pas le processeur, l'exécutif temps réel suspend la tâche pour la durée d'exécution de la procédure externe et alloue le processeur à une autre tâche prête.

Des études existent concernant les tâches à suspension. Mais ces études cherchent à construire un test d'ordonnançabilité. Souvent ces méthodes consistent à adapter des tests d'ordonnançabilité déjà existants pour l'ordonnancement de tâches périodiques aux tâches à suspension. Or ces travaux n'étudient pas directement le problème d'ordonnancement de tâches à suspension. En particulier, aucune étude n'a été menée pour, dans un premier temps, connaître les difficultés à construire un tel ordonnancement. Ainsi, par exemple, la complexité du problème d'ordonnancement des tâches à suspension n'a pas été établie, pas plus que la compétitivité des algorithmes en-ligne. Finalement, à propos des tests d'ordonnançabilité établis pour les tâches à suspension, personne ne connaît l'ampleur du pessimisme qu'ils introduisent. L'utilisation de tests trop pessimistes conduit invariablement à surdimensionner l'architecture matérielle du système temps réel. L'ensemble de ces points seront étudiés dans la suite de cette partie.

3.5. Plan de la partie

Dans le chapitre 4, nous définissons les principaux concepts de l'ordonnancement temps réel. Dans le chapitre 5, les études sur les tests d'ordonnançabilité pour l'ordonnancement de tâches à suspension sont présentées. Les chapitres suivants développent nos différentes contributions pour l'ordonnancement de tâches à suspension. Ainsi, dans le chapitre 6, les

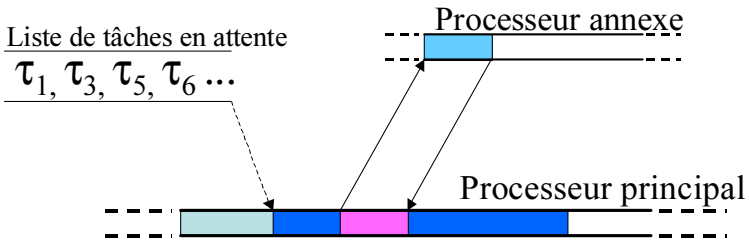

Figure 3.4. *Ordonnancement de tâches à suspension*

difficultés d'un tel ordonnancement sont exposées : la complexité, la présence d'anomalies d'ordonnancement et finalement l'optimalité des algorithmes en-ligne. Le chapitre suivant, chapitre 7, étudie la compétitivité d'algorithmes en-ligne bien connus pour deux critères d'optimisation : la minimisation du temps de réponse et la minimisation du nombre de tâches en retard. Finalement, le chapitre 8, aborde nos résultats sur la recherche du pessimisme engendré par les tests d'ordonnnaçabilité, établis pour l'ordonnancement des tâches à suspension.

3.6. Bibliographie

[BAR 90] BARUAH S., ROSIER L., HOWELL R., « Algorithms and Complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor », *Journal of Real-Time Systems*, vol. 1, p. 301-324, 1990.

[BAR 04] BARUAH S., GOOSSENS J., « Scheduling Real-time Tasks : Algorithms and Complexity », *In Handbook of Scheduling : Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (edt). Chapman Hall/CRC Press*, 2004.

[CNR 88] CNRS, « Le temps réel - Groupe de réflexion sur le temps réel », *Techniques et Sciences Informatiques*, vol. 8, n°5, p. 493–500, 1988.

[COT 05] COTTET F., GROLLEAU E., *Systèmes temps réel de Contrôle-Commande, conception et implémentation*, Dunod, Paris, 2005.

[DOR 91] DORSEUIL A., P. PILLOT, *Le temps réel en milieu industriel : Concepts, environnements, multitâches*, Dunod, Paris, 1991.

[HAR 85] HAREL D., PNUELLI A., « On the development of Reactive Systems, in Logic and Models of Concurrent Systems », *NATO ASI in Computer Science*, p. 447-498, 1985.

[KOP 83] KOPETZ H., « Real-time in Distributed Real-Time Systems », *Real-time in distributed real-time systems. Proc. 5th IFAC workshop on Distributed Computer Control Systems, Oxford, UK*, 1983.

[LAP 91] LAPLANTE P. A., « Real-Time Systems Design and Analysis, an Engineer's handbook », *IEEE Computer Society Press*, vol. 0-8186-3107-4, 1991.

[SIM 05] SIMONOT-LION F., SONG Y., *Design and validation process of in-vehicle embedded electronic systems, in The Embedded Systems Handbook*, Richard Zurawski (Edt), CRC Press - Taylor & Francis, New-York, 2005.

[STA 88] STANKOVIC J. A., « Misconception about Real-Time Systems - a serious problem for next generation systems », *IEEE Computer*, p. 10–19, October 1988.

[STA 95] STANKOVIC J., SPURI M., DINATALE M., BUTTAZZO G., « Implications of classical scheduling results for real-time systems », *IEEE Computer*, vol. 28, n°6, p. 16-25, 1995.

[TRI 05] TRINQUET Y., « Les systèmes d'exploitation temps réel », 4^{ieme} *École d'été temps réel, Nancy*, vol. 1, n°13–16, p. 123–134, septembre 2005.

Chapitre 4

Ordonnancement temps réel monoprocesseur[1]

4.1. Définitions générales

Les systèmes informatiques temps réel (ou systèmes de Contrôle-Commande) se décomposent en un exécutif temps réel et un programme à exécuter sur l'exécutif. Pour concevoir, développer et faire évoluer le système, un programme temps réel est d'un point de vue logiciel considéré comme un système *multitâche*. Le travail du système informatique consiste par conséquent à gérer l'exécution et la concurrence de l'ensemble des tâches en optimisant l'occupation du processeur et en veillant à respecter les contraintes temporelles des tâches. Toutes ces fonctions sont regroupées sous la notion d'**Ordonnancement temps réel**. Si une contrainte temporelle n'est pas respectée, on parle d'une défaillance du système ou d'une faute temporelle. Dans un système temps réel dur, chaque tâche est soumise à une échéance temporelle stricte. Le non respect d'une contrainte temporelle n'est pas admissible à l'exécution.

La périodicité de l'activation des tâches est l'unique spécificité de l'ordonnancement temps réel par rapport à la théorie de l'ordonnancement classique. Cette nuance peut paraître mineure, mais en général la périodicité des tâches change la nature combinatoire des problèmes d'ordonnancement. Toutefois, l'ordonnancement temps réel nécessite de recourir aux mêmes techniques algorithmiques de résolution de problèmes combinatoires [STA 95].

4.1.1. *Tâche temps réel*

Une tâche temps réel est l'entité de base des programmes temps réel. Ce sont ces entités qui composent le programme d'un système temps réel. Les tâches temps réel peuvent être associées à des calculs, des alarmes, des traitements d'entrée/sortie, etc.

Les tâches temps réel permettent ainsi de contrôler le procédé externe via un ensemble de capteurs et d'actionneurs intégrés au procédé. Elles utilisent les primitives de l'exécutif

1. Ce chapitre est tiré du livre d'Hermès intitulé *Systèmes Temps Réel 2 : Ordonnancement, Réseaux et Qualité de Service* réalisé sous la direction de Nicolas Navet (chapitre : "ordonnancement monoprocesseur") et édité par Hermes Science Publications en 2006

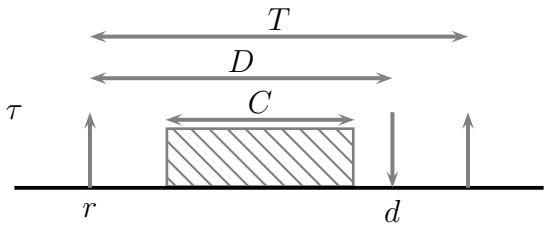

Figure 4.1. *Le modèle des tâches périodiques*

temps réel sur lequel elles sont exécutées pour communiquer entre elles, faire des acquisitions de données (depuis des capteurs du système contrôlé) ou encore actionner des commandes sur le système contrôlé. Mais pour respecter les caractéristiques du procédé (p. ex. désactiver à temps le contrôleur de vitesse d'une voiture dont le conducteur actionne les freins), les tâches sont soumises à des contraintes temporelles. Il existe plusieurs modèles de tâches : les tâches *périodiques*, *apériodiques* et *sporadiques*. Une *configuration* de tâches est définie par n ($n \in \mathbb{N}^*$) tâches.

4.1.1.1. *Tâches périodiques*

Les tâches périodiques représentent les tâches récurrentes dont les activations successives sont séparées par une période constante. Il existe dans la littérature temps réel plusieurs modèles de tâches périodiques. Le plus simple mais aussi le plus fondamental est celui de LIU et LAYLAND [LIU 73]. Leur modèle représente chaque tâche par deux paramètres : $\tau = (C, T)$, où C désigne le temps d'exécution maximum de la tâche et T sa période. Par conséquent, toutes les T unités de temps, la tâche τ génère une **requête** (la première requête étant générée à l'instant 0) qui doit s'exécuter au plus pendant C unités de temps avant son échéance qui coïncide avec la date d'activation de la prochaine requête de la tâche. D'autres modèles existent mais les principaux, basés sur le modèle de LIU et LAYLAND, ne font que rajouter des paramètres au modèle de base.

Le paramètre r permet de retarder la première exécution des tâches et n'impose pas ainsi aux tâches de commencer leur exécution au même moment à un instant 0. Ce paramètre est aussi appelé *Date d'activation* de la tâche. La date d'activation de la k^{e} requête d'une tâche est égale à $r + (k-1)T$. Deux tâches τ_1 et τ_2 sont à départ simultané (ou synchrones) si leurs dates d'activation sont égales (c.-à-d. $r_1 = r_2$).

Le paramètre d dénote la date d'échéance de la tâche, date à partir de laquelle, la tâche doit avoir terminé son exécution. Ce paramètre permet de différencier l'échéance d'exécution des requêtes de la date d'activation de la prochaine requête. À partir de cette échéance (dite absolue) de la k^{e} requête, il est possible de définir une autre échéance, l'échéance relative (par rapport à l'activation de la k^{e} requête) $D = d-(k-1)T$ de la tâche. L'échéance relative est aussi appelée *délai critique*. Si l'échéance relative D d'une tâche τ est égale à sa période T, la tâche est dite à *échéance sur requête*. La figure 4.1 représente les paramètres de chaque tâche τ périodique.

Des caractéristiques supplémentaires peuvent être calculées pour chaque tâche périodique :

– Le facteur d'utilisation du processeur par une tâche τ : $u = \frac{C}{T}$. Le facteur d'utilisation d'un ensemble de tâches $I = \{\tau_i\}_{1 \leq i \leq n}$ peut ainsi se définir comme la somme des facteurs d'utilisation des tâches τ_i : $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$.

– Le facteur de charge du processeur par une tâche τ : $u_l = \frac{C}{D}$. Le facteur de charge d'un ensemble de tâches $I = \{\tau_i\}_{1 \leq i \leq n}$ peut ainsi se définir comme la somme des facteurs de charge des tâches τ_i : $U_l = \sum_{i=1}^{n} \frac{C_i}{D_i}$.

– La laxité dynamique d'une tâche τ représente le nombre d'unités de temps que le processeur peut passer à ne pas exécuter τ sans risquer de manquer son échéance. La laxité dynamique d'une tâche τ à l'instant t : $L(t) = D(t) - C(t)$ où $D(t) = d - t$ et $C(t)$ le temps d'exécution restant à l'instant t.

– Le temps de réponse de la k^{e} requête d'une tâche : R^k, est la différence de temps entre l'instant où la tâche termine son exécution et l'instant où elle s'est activée.

– Le temps de réponse d'une tâche : R, est le maximum des temps de réponse de ses requêtes $R = \max_{k \geq 1} R^k$.

Pour déterminer l'ordonnancement des tâches d'une configuration, les algorithmes affectent suivant leurs critères une priorité à chaque tâche (notée $Prio$). La tâche ayant la plus haute priorité est exécutée. Nous considérerons dans la suite, que la tâche la plus prioritaire est celle dont la valeur de $Prio$ est la plus faible.

Avant de commencer son exécution, il est possible qu'une tâche doive attendre la fin de l'exécution d'une autre tâche. En effet, des tâches peuvent avoir entre elles, des relations de synchronisation (sémaphore, l'activation d'un évènement attendu par une autre, rendez-vous, communication). Lorsque deux tâches sont ainsi liées, elles sont dites *dépendantes* (sinon, elles sont *indépendantes*) et possèdent une *contrainte de précédence*. Il existe deux types de contraintes de précédence : les contraintes simples et les contraintes généralisées. Une contrainte de précédence simple entre deux tâches τ_i et τ_j (c.-à-d. τ_i précède τ_j) est définie si la tâche τ_j doit attendre la fin de l'exécution de la tâche τ_i pour débuter la sienne. Les deux tâches ont alors la même période et un même nombre d'exécutions. Une contrainte de précédence est généralisée si le nombre des exécutions des deux tâches n'est pas le même (c.-à-d. τ_i s'exécute plusieurs fois avant que τ_j s'exécute ou τ_i s'exécute une fois avant que τ_j s'exécute plusieurs fois) [RIC 01].

Des tâches peuvent au cours de leur exécution, partager des ressources critiques en exclusion mutuelle (c.-à-d. qu'à chaque instant, au plus une seule tâche peut utiliser la ressource). Une tâche ne peut pas utiliser une ressource occupée par une autre tâche.

4.1.1.2. *Tâches apériodiques*

Une tâche apériodique a une date d'activation qui n'est connue qu'à son activation et une certaine durée d'exécution. Par conséquent, deux paramètres suffisent pour représenter une tâche apériodique : $\tau = (r, C)$, où r est sa date d'activation et C son temps d'exécution. Les tâches apériodiques sont en général à contraintes souples car elles n'ont pas d'échéance avant la fin de leur exécution. Dans le cas contraire, un paramètre d'échéance d sera ajouté dans le modèle des tâches.

4.1.1.3. *Tâches sporadiques*

Les tâches sporadiques sont des tâches récurrentes avec des contraintes strictes d'échéance. Une tâche sporadique est définie par un paramètre C et une durée minimum T entre deux requêtes successives de la tâche. Le modèle peut être généralisé de la même façon que le modèle de tâche périodique.

4.1.2. *Ordonnancement*

Le concepteur d'un système temps réel définit une configuration (ou système) de tâches périodiques. Elles sont soumises à des contraintes temporelles dont le respect doit être validé (vérifié) avant la mise en service du système ou durant l'exécution des tâches si leurs caractéristiques ne sont pas connues a priori.

4.1.2.1. *Algorithmes hors-ligne/en-ligne*

Les algorithmes d'ordonnancement peuvent être classés en deux catégories, les algorithmes d'ordonnancement hors-ligne et en-ligne.

Les *algorithmes d'ordonnancement hors-ligne* construisent la séquence d'ordonnancement complète sur les bases des paramètres temporels de l'ensemble des tâches de la configuration. L'ordonnancement est construit complètement avant l'exécution de l'application.

Les *algorithmes d'ordonnancement en-ligne* choisissent, quant à eux, dynamiquement la prochaine requête à exécuter en fonction des paramètres des tâches prêtes à l'exécution. Ils ne connaissent à un instant t que les tâches dont les dates d'activation sont antérieures à t.

Un algorithme d'ordonnancement en-ligne est *préemptif* si l'algorithme peut préempter l'exécution d'une tâche pour en choisir une autre. Dans le cas inverse, l'algorithme est *non préemptif*.

Selon le type d'attribution des priorités aux tâches, il existe trois *classes* d'algorithmes en-ligne :

– Les *algorithmes à priorité fixe pour les tâches [LIU 73]* attribuent une priorité à chaque tâche (priorité qu'elle gardera définitivement) pendant la conception du système. Ainsi chaque requête d'une tâche hérite de la priorité de la tâche.

– Les *algorithmes à priorité fixe pour les requêtes* peuvent assigner une priorité différente à chaque requête d'une tâche. Ainsi, dès qu'une nouvelle requête de tâche arrive dans le système, elle pourra avoir une priorité différente de celle de la requête précédente de la même tâche mais la gardera inchangée pendant toute la durée de la requête.

– Les *algorithmes à priorité dynamique* assignent une priorité aux tâches mais les priorités peuvent évoluer à chaque instant. Une requête peut ainsi voir varier sa priorité durant son exécution.

Un algorithme d'ordonnancement en-ligne est *conservatif*, s'il ne laisse jamais le processeur inoccupé lorsque des tâches sont prêtes à s'exécuter.

Un algorithme A est *optimal*, si toute configuration pouvant être ordonnancée en respectant toutes les contraintes temporelles des tâches l'est avec l'algorithme A. Et inversement, si A ne peut pas ordonnancer une configuration de ce type alors aucun algorithme ne peut l'ordonnancer.

4.1.2.2. *Validation et Ordonnancement*

La validation et l'ordonnancement d'un système sont très souvent considérés comme des problèmes disjoints, bien qu'ils soient bien sûr fortement imbriqués en réalité. Lorsque l'ordonnancement est construit durant l'exécution (c.-à-d. en-ligne), alors la validation consiste à analyser le comportement d'un algorithme fondé en général sur l'attribution de priorités aux tâches. L'ordonnancement et la validation sont alors clairement séparés et n'interagissent pas (c.-à-d. pas d'échange de données entre les étapes de validation et d'ordonnancement). Cette décomposition du problème d'ordonnancement consistant d'une part, à choisir un algorithme d'ordonnancement, et d'autre part, à analyser son comportement, permet de réduire la validation à un problème d'évaluation des performances. L'algorithme de validation est désigné sous le nom de **test d'ordonnançabilité** dans la littérature [BAR 04]. Un test positif indique que pour tous les comportements possibles du système alors les contraintes temporelles des tâches seront respectées durant leur exécution.

4.1.2.3. *Simulation/Analyse pire cas*

Les tâches périodiques sont exécutées indéfiniment. Bien que l'ordonnancement soit infini, la périodicité des tâches permet de limiter la recherche des fautes temporelles dans un intervalle de temps borné, appelé intervalle de faisabilité ou d'étude (*feasibility interval*) [LEU 80]. La périodicité de l'ordonnancement dépend du *ppcm*[2] des périodes des tâches et est désignée sous le nom d'*hyperpériode* :

– la longueur de l'intervalle d'étude est égale au ppcm des périodes si les tâches sont à départ simultané ;

– si les tâches ne sont pas à départ simultané, la longueur de l'intervalle d'étude est égale au maximum des dates d'activation plus deux fois le ppcm des périodes : $\max_i\{r_i\} + 2\ ppcm\{T_i\}$.

Pour des problèmes plus complexe, il est nécessaire de montrer un résultat équivalent pour montrer :

– le système entre régime périodique (p. ex. ppcm dans le contexte de [LEU 80])

– définir la longueur de régime transitoire (p. ex. $\max r_i + ppcm$, dans le contexte de [LEU 80]).

Simplifier le test d'ordonnançabilité en un problème d'évaluation de performance suggère l'utilisation de techniques de simulation. Bien que cette technique soit très utile durant la conception préliminaire d'un système, nous rejetons son utilisation comme test d'ordonnançabilité pour les stratégies d'ordonnancement en-ligne. En effet, l'idée couramment rencontrée, consistant à simuler le comportement du système sur les bases des caractéristiques

2. Plus Petit Commun Multiple

tâches	r_i	C_i	D_i	T_i	$Prio_i$
τ_1	0	1	3	**3**	**1**
τ_2	0	4	6	6	2

Tableau 4.1. *Exemple d'affectation de priorités selon* RM

de l'ordonnanceur et du comportement pire cas de chaque tâche, ne conduit pas nécessairement au pire comportement du système. L'instabilité d'un ordonnancement, liée à la variation des paramètres des tâches durant leur exécution, peut engendrer des situations qui n'auront jamais été simulées. La simulation, aussi bien que les tests logiciels classiques, ne permettent en aucun cas de conclure avec certitude que les tâches respecteront leurs contraintes temporelles. Seule une analyse pire cas, qui repose sur la caractérisation du pire comportement de l'application temps réel, permettra d'arriver à une conclusion fiable, sous réserve du respect des hypothèses.

4.2. Principaux algorithmes d'ordonnancement préemptif en-ligne

Dans ce paragraphe, nous détaillons les principaux algorithmes classiques en-ligne pour l'ordonnancement préemptif de tâches périodiques. Pour chacun de ces algorithmes, le mode d'affectation des priorités aux tâches ainsi que son fonctionnement sont détaillés.

4.2.1. *Rate Monotonic (*RM*)*

L'algorithme à priorité fixe pour les tâches, *Rate Monotonic* ou (RM), a été introduit par LIU et LAYLAND en 1973 [LIU 73].

Définition 8 *Les priorités selon l'algorithme* RM *sont dans l'ordre inverse des périodes : la tâche de plus faible période a la plus grande priorité et la tâche de plus grande période a la plus faible priorité.*

Exemple 1 *Affectation de priorités et d'ordonnancement d'une configuration I sous* RM *: le tableau 4.1 représente l'affectation de priorités pour une configuration à deux tâches. τ_1 est la tâche ayant la plus petite période, elle se retrouve donc avec la plus grande priorité. Ainsi comme l'illustre la figure 4.2, la tâche τ_1 est exécutée avant τ_2.*

Propriété 1 *L'algorithme* RM *est optimal dans la classe des algorithmes à priorité fixe pour l'ordonnancement de tâches périodiques (ou sporadiques), indépendantes, à départ simultané au démarrage et à échéance sur requête. Si l'une de ces deux dernières conditions est relâchée,* RM *n'est plus optimal.*

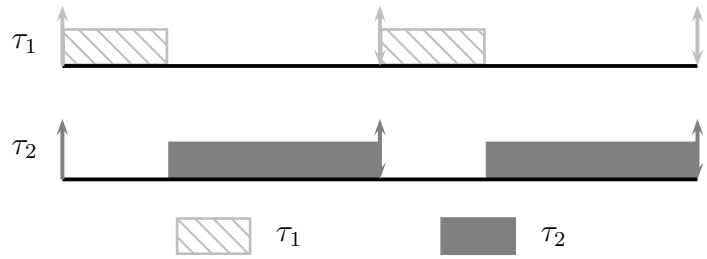

Figure 4.2. *Ordonnancement de la configuration du tableau 4.1 avec l'algorithme* RM

tâches	r_i	C_i	D_i	T_i	$Prio_i$
τ_1	0	1	**4**	4	**1**
τ_2	0	3	6	6	2
τ_3	0	2	8	8	3

Tableau 4.2. *Exemple d'affectation de priorités selon* DM

4.2.2. *Deadline Monotonic (*DM*)*

L'algorithme *Deadline Monotonic* ou (DM) a été introduit par LEUNG et WHITEHEAD en 1982 [LEU 82]. Cet algorithme, à priorité fixe pour les tâches, diffère de *Rate Monotonic* car il base son ordonnancement non pas sur les périodes mais sur les échéances relatives.

Définition 9 *Les priorités selon l'algorithme Deadline Monotonic sont dans l'ordre inverse des échéances relatives : la tâche de plus faible échéance relative a la plus grande priorité et la tâche de plus grande échéance relative a la plus faible priorité.*

Exemple 2 *Affectation des priorités d'une configuration selon* DM *:*
le tableau 4.2 montre l'affectation de priorités pour une configuration à trois tâches. La plus haute priorité est affectée à τ_1 car elle est la tâche avec la plus faible échéance relative.

Propriété 2 *L'algorithme* DM *est optimal pour l'ordonnancement de tâches périodiques (ou sporadiques), indépendantes et à départ simultané au démarrage. À noter que si les tâches sont à échéance sur requête,* RM *et* DM *sont équivalents.*

4.2.3. *Earliest Deadline First (*EDF*)*

Earliest Deadline First ou (EDF), qui est un algorithme à priorité fixe pour les requêtes, fut présenté par *Jackson* en 1955 [JAC 55].

Définition 10 *Earliest Deadline First attribue, à chaque instant t, la priorité la plus grande à la requête ayant la plus petite échéance absolue.*

Exemple 3 *Ordonnancement d'une configuration de tâches sous* EDF. *La configuration est celle présentée pour* DM *(Tableau 4.2) :*
la longueur de l'ordonnancement à tester est de 24 unités de temps (24 = ppcm(4,6,8)) (cf. Paragraphe 4.1.2.3). Cette configuration n'est pas ordonnançable sous DM *car la tâche* τ_3 *manque sa première échéance. Cette configuration est ordonnançable par* EDF *: à l'instant* 0, EDF *affecte les mêmes priorités que* DM *(tableau 4.2). Par conséquent, jusqu'à l'instant* 6, *les ordonnancements sous* EDF *et sous* DM *sont identiques. Mais à cet instant, les priorités sous* EDF *changent alors qu'elles restent statiques avec* DM. *Ainsi la requête* $\tau_{3,1}$ *récupère la plus haute priorité et s'exécute avant* $\tau_{2,2}$.

Propriété 3 *Earliest Deadline First est optimal pour l'ordonnancement de configuration de tâches indépendantes. Par conséquent, si une configuration (avec les caractéristiques précédentes) est ordonnançable, elle le sera par* EDF *[DER 74, LAB 74].*

4.2.4. *Least Laxity First (*LLF*)*

Least Laxity First [MOK 83] est un algorithme à priorité dynamique. Les priorités assignées par LLF aux requêtes sont dans l'ordre inverse des valeurs de la laxité dynamique des requêtes.

Définition 11 *Least Laxity First attribue à chaque instant* t *la priorité la plus élevée à la requête ayant la plus faible laxité dynamique. La laxité de la* k^e *requête de* τ_i *à un instant* t *est égale à* $L_{i,k}(t) = d_{i,k} - t - C_i(t) = D_i(t) - C_i(t)$, *où* $d_{i,k}$ *est l'échéance absolue de la* k^e *requête.*

Exemple 4 *Ordonnancement d'une configuration de tâches sous* LLF. *Cette configuration est celle choisie pour montrer un exemple d'affectation de priorités par* DM *(Tableau 4.2) :*
Elle est non ordonnançable par DM. *L'ordonnancement sous* LLF *est lui faisable : à l'instant* 0, LLF *affecte les mêmes priorités aux tâches que* DM *(tableau 4.2). Mais comme* LLF *est un algorithme à priorité dynamique, à l'instant* 4, $\tau_{3,1}$ *devient plus prioritaire que* $\tau_{1,2}$. *Et à l'instant* 5, *inversement,* $\tau_{1,2}$ *est plus prioritaire que* $\tau_{3,1}$. *Ainsi l'ordonnancement est faisable sous* LLF.

Propriété 4 *Least Laxity First (comme* EDF*), est un algorithme optimal pour l'ordonnancement de tâches indépendantes et avec des échéances relatives inférieures ou égales aux périodes [DER 89, MOK 83].*

4.3. Extension de l'ordonnancement

Dans ce paragraphe, nous présentons plusieurs extensions classiques de l'ordonnancement temps réel : l'ordonnancement de tâches avec gigue sur activation, surcharge du processeur, ordonnancement en présence de tâches apériodiques, les anomalies d'ordonnancement, les contraintes de précédence et le partage de ressources critiques.

4.3.1. *Gigue sur activation*

En réalité, entre l'instant d'activation d'une tâche périodique et l'instant où elle est prête à s'exécuter, il peut exister un délai variable appelé aussi *gigue sur activation*. Ce délai noté J_i pour la tâche τ_i, permet de représenter le temps nécessaire pour effectuer l'acquisition de données (réseau ...). La prise en compte de ce paramètre supplémentaire nécessite d'adapter les tests d'ordonnnançabilité. On pourra par exemple consulter [AUD 93, TIN 92, BUR 95] pour les algorithmes à priorité fixe et [SPU 96a] pour les algorithmes à priorité dynamique.

4.3.2. *Surcharge du processeur*

Un système est en surcharge processeur lorsque les ressources nécessaires pour exécuter les tâches du système sont plus importantes que les ressources disponibles.

L'algorithme d'ordonnancement EDF est très sensible aux surcharges du processeur. En effet, si une tâche vient à manquer son échéance, EDF qui fonde ses décisions d'ordonnancement sur la proximité de l'échéance des tâches, donnera ainsi plus grande priorité à la tâche qui est en train de manquer son échéance. Ainsi les autres tâches qui suivent la tâche en retard se retrouveront elles-mêmes retardées et manqueront à leur tour leur échéance. Cette escalade de fautes temporelles est appelée *l'effet domino* [STA 95]. Des techniques d'augmentation de ressources (rajouter des processeurs ou augmenter la vitesse du processeur) permettent d'améliorer l'efficacité des algorithmes d'ordonnancement en-ligne et plus particulièrement EDF [CHA 05, PHI 97] dans les situations de surcharge.

4.3.3. *Ordonnancement de configuration comprenant des tâches apériodiques*

Lors de l'ordonnancement de configuration de tâches périodiques, des tâches apériodiques (p. ex. des alarmes du système) peuvent apparaître et doivent être ordonnancées sur le processeur. Il existe plusieurs méthodes pour l'ordonnancement de tâches apériodiques sous RM et EDF. Une première méthode (*en arrière plan*) consiste à ordonnancer les tâches apériodiques pendant les temps de oisiveté du processeur. Une seconde approche consiste à créer une tâche périodique dédiée au service des tâches apériodiques. Plusieurs types de serveurs ont été proposés dans la littérature : le serveur à scrutation (*polling server*) [SPR 89], serveur ajournable (*Deferrable server*) [LEH 87, STR 95], serveur à échange de priorités (*Priority Exchange server*) [SPR 88, LEH 87, SPU 94], serveur sporadique (*Sporadic server*) [SPR 89, SHA 88, SPU 94, SPU 96b] et le serveur *Earliest Deadline Last* (*EDL*) [CHE 89]. Enfin, dernier type de serveur, les serveurs à largeur de bande : *Constant Bandwidth Server* (*CBS*) présenté par *Abeni* [ABE 98] et *Total Bandwidth Server* (*TBS*) présenté par SPURI et BUTTAZZO [SPU 94, SPU 95]. Nous renvoyons aux références pour la définition de ces différentes approches.

4.3.4. *Anomalies d'ordonnancement*

Une anomalie d'ordonnancement est liée à l'instabilité de l'ordonnanceur. Ce problème est bien connu en environnement multiprocesseur. Nous utilisons la défintion de [GRA 69], qui

Tâches	C_i	$D_i = T_i$	$Prio_i$
τ_1	1	3	1
τ_2	2	6	2
τ_3	4	12	3

Tableau 4.3. *Tâches à échéance sur requête à ordonnancer sans préemption et avec des priorités fixes ($Prio_i$)*

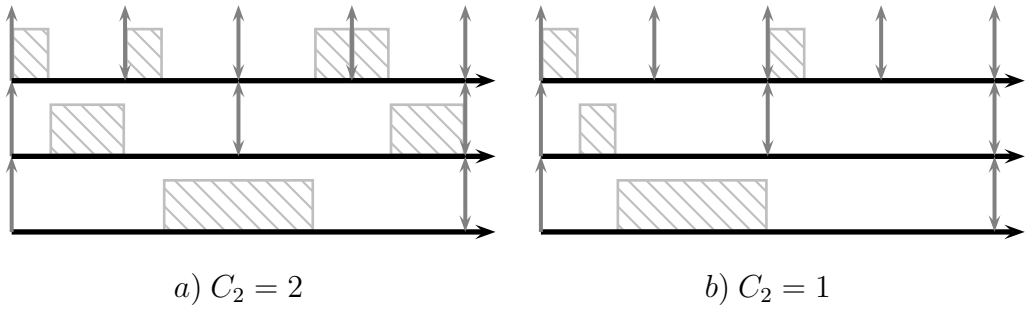

Figure 4.3. *Anomalies d'ordonnancement en non préemptif pour le système de tâches du tableau 4.3*

se formule de la façon suivante : *un système ordonnançable avec les pires durées d'exécution peut être non ordonnançable avec des durées d'exécution plus petites*. Dans la littérature, il existe d'autres anomalies d'ordonnancement comme par exemple, les anomalies d'ordonnancement qui surviennent lorsque la période des tâches est augmentée [AND 03, NAV 06]. Mais, dans la suite, nous nous intéressons uniquement à la définition présentée par GRAHAM [GRA 69]

La validation d'algorithme d'ordonnancement basé sur les priorités est un problème difficile. Le temps d'exécution d'une tâche peut varier d'une requête à l'autre et il faut vérifier la validation du système pour toutes les variations possibles. Mais si un système ne possède pas d'anomalie d'ordonnancement, la validation du système peut se simplifier en considérant pour durée d'exécution de chaque requête, la pire durée d'exécution de la tâche. Le système est alors *robuste* (il n'est pas sujet à des anomalies d'ordonnancement pour le système de tâches considéré). Une *anomalie d'ordonnancement* survient lorsque réduire la durée d'exécution d'une tâche peut augmenter le temps de réponse de certaines tâches pouvant ainsi conduire à manquer une échéance. Voici un exemple de système monoprocesseur qui subit des anomalies d'ordonnancement :

Exemple 5 *Sur le tableau 4.3 est représenté un système de tâches à ordonnancer sans préemption. La figure 4.3 présente deux ordonnancements : dans le premier, les tâches sont exécutées avec leurs pires durées d'exécution (a), tandis que dans le second, la tâche* 2 *s'exécute en* 1 *seule unité de temps au lieu de* 2 *(b). Ainsi, en construisant la même séquence de requêtes, réduire la durée d'exécution de τ_2 rend le système non ordonnançable.*

Voici d'autres exemples de système monoprocesseur subissant des anomalies d'ordonnancement : les système préemptifs avec algorithme à priorité fixe et tâches avec contraintes de

précédence (une autre anomalie survient si le nombre de contraintes de précédence diminue). Un autre exemple, celui des systèmes préemptifs avec algorithme dynamique (*EDF*) et tâches à suspension [RID 04]. Les tâches à suspension sont des tâches qui au court de leur exécution ont besoin de se suspendre pour effectuer des opérations externes (telles que des opérations d'entrée/sortie).

4.3.5. *Contraintes de précédence*

Dans ce paragraphe, nous détaillons les solutions existantes dans la littérature pour l'ordonnancement de configurations de tâches avec contraintes de précédence.

Sous EDF, l'ordonnancement de tâches soumises à des contraintes de précédence peut se faire en modifiant les dates d'arrivée et les échéances relatives des requêtes des tâches [CHE 90, SPU 93, JEF 92]. L'ordonnancement construit par EDF conduit à respecter les contraintes de précédence et les échéances. Une telle approche ne peut pas être effectuée pour les algorithmes à priorité fixe en raison d'existence d'anomalies d'ordonnancement [RIC 02].

4.3.6. *Partage de ressources*

Lorsqu'un ensemble de tâches partage des ressources critiques en exclusion mutuelle, il est prouvé qu'il n'est pas possible de déterminer un algorithme en-ligne optimal [BAR 90b, LEU 80, LEU 82]. Il existe deux principaux problèmes : l'*inversion de priorité* qui survient lorsqu'une tâche possède une ressource, bloquant ainsi une autre tâche plus prioritaire utilisant la même ressource et laisse s'exécuter des tâches de priorité intermédiaire mais n'utilisant pas la ressource et l'*interblocage* qui survient lorsque deux tâches ayant besoin de deux mêmes ressources critiques pour leur exécution en possèdent chacune une. Ainsi, elles se retrouvent mutuellement bloquées dans l'attente de la libération de la ressource manquante.

Des protocoles de gestion de ressources ont été créés pour pallier ces problèmes et ainsi améliorer l'ordonnancement des tâches : le Protocole à Héritage de Priorité (*Priority Inheritance protocol*) [RAJ 91, SHA 90] et le Protocole à Priorité Plafond (*Priority Ceiling protocol*) [RAJ 89, SHA 90] et *SRP* (*Stack Resource Protocol*) pour EDF [BAK 91].

4.4. Introduction aux tests d'ordonnançabilité

Nous considérons la validation d'un système reposant sur un ordonnanceur en-ligne. Trois techniques d'analyse pire cas existent dans la littérature (tests d'ordonnançabilité) : **l'analyse du facteur d'utilisation du processeur, l'analyse de la demande processeur et l'analyse du temps de réponse**. Ces techniques ne sont pas en pratique équivalentes.

La majorité des monographies en temps réel présentent séparément l'ordonnancement des tâches à priorité fixe et à priorité dynamique. Nous choisissons ici une présentation différente afin de clairement dissocier la différence entre les tests exacts et approchés. **Nous nous limitons aux tests sur les tâches périodiques**.

4.4.1. *Principe des analyses d'ordonnançabilité*

Les tests d'ordonnançabilité établis pour chaque analyse peuvent se classer en deux catégories : les tests exacts et les tests approchés. Les tests exacts fournissent des conditions nécessaires et suffisantes de garantie du respect des échéances des tâches (pas toujours des approches pire cas). Les tests approchés ne sont que des conditions suffisantes d'ordonnançabilité. Ils sont soit des approximations de tests exacts mais avec une complexité algorithmique moindre, soit ils ne permettent pas de conclure pour toutes les configurations de tâches. Dans ce dernier cas, la réponse des tests d'ordonnançabilité retourne *oui*, la configuration de tâches est ordonnançable ou dans le cas inverse, le test ne peut pas conclure.

L'analyse du facteur d'utilisation cherche à déterminer si les besoins de la configuration de tâches à ordonnancer n'excèdent pas les capacités du processeur.

Définition 12 *L'analyse du facteur d'utilisation est un test d'ordonnançabilité qui consiste à calculer le facteur d'utilisation d'une configuration de tâches puis à vérifier qu'il n'excède pas la valeur seuil propre à l'algorithme d'ordonnancement utilisé.*

Dans le principe, l'analyse du temps de réponse et l'analyse de la demande processeur repose sur un même constat : **un dépassement d'échéance ne survient jamais lorsque le processeur est libre**. Ces analyses se restreignent à l'étude des intervalles de temps où le processeur exécute des tâches : **les périodes d'activité (busy periods)** [LEH 89]. L'analyse du temps de réponse et l'analyse de la demande processeur portent sur l'étude d'une ou plusieurs périodes d'activité. Chaque période d'activité à analyser va être caractérisée par **un scénario pire cas**. Bien que les deux analyses reposent sur le même constat, les principes de ces deux tests sont différents.

Définition 13 *[JOS 86] L'analyse du temps de réponse est un test d'ordonnançabilité en deux étapes :*

– tout d'abord le pire temps de réponse R_i (ou une borne supérieure) de chaque tâche est calculé,

– puis le respect des échéances est testé en vérifiant : $R_i \leq D_i, 1 \leq i \leq n$ (complexité algorithmique en $O(n)$).

L'analyse des temps de réponse se fait tâche par tâche et la complexité du test d'un système de tâches est principalement liée au calcul des pires temps de réponse. Par contre, l'analyse de la demande processeur est un test considérant toutes les tâches simultanément.

Définition 14 *L'analyse de la demande processeur revient à tester pour tout intervalle de temps $[t_1, t_2]$ que la durée maximum cumulée (ou une borne supérieure) des exécutions des requêtes, qui ont leur réveil et leur échéance dans l'intervalle, est inférieure à $t_2 - t_1$ (c.-à-d. n'excède pas la longueur de l'intervalle).*

4.4.2. *Définition de principaux concepts*

Nous présentons maintenant les concepts nécessaires à l'étude de ces techniques d'analyse de l'ordonnançabilité d'un système de tâches. Nous ne considérons, dans la suite, que des tâches périodiques indépendantes.

4.4.2.1. *Périodes d'activité et scénario de test*

Définition 15 *Une période d'activité du processeur est un intervalle de temps $]a, b[$ de l'ordonnancement tel que le processeur a exécuté toutes les requêtes arrivées avant la date a et a terminé à la date b toutes les requêtes arrivées à partir de la date a.*

Lorsque l'ordonnanceur est conservatif, c'est-à-dire qu'il n'insère pas de temps creux dans l'ordonnancement s'il existe une tâche prête à s'exécuter, le nombre de périodes d'activités différentes est fini puisque l'ordonnancement est périodique avec une période égale au ppcm des périodes des tâches [LEU 80]. Précisément, l'ordonnanceur va être confronté exactement au même scénario d'activation des tâches à chaque début d'hyperpériode et prendra en conséquence exactement les mêmes décisions. Toutefois, l'analyse de toutes les périodes d'activité n'est en général pas possible puisque leur nombre est exponentiel. De plus, trouver dans quelle période d'activité une tâche ne respectera pas son échéance n'est pas un problème simple et nécessite dans la majorité des cas un temps de calcul exponentiel dans la taille du système de tâches à analyser. Pour des systèmes simples de tâches, nous allons présenter les résultats analytiques connus pour caractériser la période d'activité et trouver les tâches ne respectant pas leurs échéances dans un système non ordonnançable.

Dans le cas d'un système à priorité fixe, tester l'ordonnançabilité d'une tâche τ_i ne nécessite pas de considérer les tâches moins prioritaires puisqu'elles n'engendreront pas d'interférence sur τ_i. Ceci revient à définir une période d'activité se limitant à un sous-ensemble de tâches prioritaires.

Définition 16 *(Systèmes à priorité fixe) Une période d'activité du processeur de niveau i est un intervalle de temps où le processeur n'exécute que des tâches ayant une priorité supérieure ou égale à i (i-level busy period).*

Une période d'activité va être caractérisée par les dates d'activation de requêtes qui la débutent. Nous pouvons maintenant définir la notion de scénario.

Définition 17 *Un scénario est l'ensemble des dates d'activation des requêtes permettant de caractériser* une *période d'activité du processeur.*

Deux cas se produiront suivant que les pires scénarios considérés durant le test se produiront ou non durant la vie du système :

– le scénario se produit nécessairement dans la vie du système : le test d'ordonnançabilité résultant est alors exact et définit une condition nécessaire et suffisante pour que le système de tâches soit ordonnançable,

– le scénario ne se produit pas forcément dans la vie du système : le test est alors approché puisque la demande processeur est surestimée. Ceci introduit donc du *pessimisme* dans le test d'ordonnançabilité, c'est-à-dire qu'une borne supérieure de la demande processeur est calculée. Le test est uniquement une condition suffisante d'ordonnançabilité : si le test renvoie vrai alors le système est ordonnançable, sinon on ne peut pas conclure.

Exemple 6 *Quelques exemples de scénarios conduisant soit à un test exact, soit à un test approché :*

• *Tests exacts : le scénario se produit toujours dans la vie de l'application.*

- Ordonnancement de tâches périodiques à départ simultané et à priorité fixe : le pire scénario est défini par le réveil simultané d'une requête de chaque tâche (c.-à-d. l'instant critique). Ce scénario se produit au démarrage de l'application.

- Ordonnancement de tâches à départ différé et gigue sur activation. On peut toujours définir les valeurs des gigues de façon à créer un instant critique (resynchroniser l'activation des tâches au début d'une période d'activité) et se ramener ainsi au pire scénario développé ci-dessus.

• *Tests approchés : le scénario peut ne pas se produire dans la vie de l'application.*

- Ordonnancement de tâches périodiques à départ différé et à priorité fixe : le pire scénario est défini par le réveil simultané des tâches. Savoir si ce scénario se produira dans la vie de l'application est co-NP-Complet (problème de congruences simultanées). En conséquence, la demande processeur calculée sera une borne supérieure.

- Ordonnancement de tâches périodiques à priorité fixe en non préemptif : le pire scénario pour la tâche τ_i survient lorsqu'elle est réveillée en même temps que les requêtes des tâches plus prioritaires ($\tau_1, ..., \tau_{i-1}$) et juste après de la plus longue tâche parmi les moins prioritaires ($\max_{j=i+1,...,} C_j$). Analyser si ce scénario surviendra ou non dans la vie de l'application n'est pas un problème simple a priori. En conséquence le test correspondant sera approché. Récemment, [BRI 06] montre que le calcul du temps de réponse était optimiste (c.-à-d. faux) et que le pire temps de réponse n'est pas toujours associé à la première requête au sein de la période d'activité définie dans ce scénario.

Afin de faciliter la conception de tests exacts, deux hypothèses se rencontrent presque systématiquement dans la littérature. Elles permettent d'assurer que le pire scénario se produit dans la vie du système. Ces hypothèses sont d'introduire une *gigue sur activation* (permettant ainsi de traiter les systèmes distribués) ou de considérer des *tâches sporadiques* plutôt que des tâches périodiques (la période entre deux requêtes est une durée minimum et non une durée exacte, comme dans le cas des tâches périodiques). Ces hypothèses supplémentaires permettent de considérer un problème d'ordonnancement plus général, mais dont les pires scénarios sont plus simples à caractériser, en ordonnancement monoprocesseur.

Exemple 7 *Voici des exemples introduisant une simplification en jouant sur la périodicité des activations des tâches :*

– Ordonnancement de tâches sporadiques à priorité fixe et à départ différé : le scénario où toutes les tâches se réveillent simultanément pourra se produire dans la vie du système. Dans le cas de tâches strictement périodiques, une telle caractérisation n'existe pas.

– Ordonnancement EDF *de tâches sporadiques : le pire temps de réponse d'une tâche τ_i survient dans une période d'activité où toutes les tâches autres que τ_i sont réveillées simultanément et où τ_i se réveille à une date d'échéance d'une autre tâche. Le test exact de Spuri [SPU 96a] dans le cas sporadique ne serait pas exact si les tâches étaient strictement périodiques, car le pire scénario ne se produirait pas forcément dans la vie du système.*

– Ordonnancement de tâches périodiques à priorité fixe et départ différé avec gigue sur activation des tâches : les gigues vont permettre de resynchroniser les tâches de façon à définir le pire scénario qui se produira dans la vie du système.

4.4.2.2. *Évaluation de la demande processeur*

Caractériser l'activité du processeur revient à compter les requêtes activées dans un intervalle de temps (c.-à-d. dans la période d'activité). Deux fonctions vont être particulièrement utiles pour analyser l'activité du processeur associée aux exécutions des requêtes des tâches. Cette activité sera dans la suite désignée sous le nom de *demande processeur*. Deux fonctions permettent de définir la demande processeur sur un intervalle de temps [MAN 98, BAR 03] :

– la demande processeur des tâches réveillées avant une date t, qui sera notée $rbf(t)$ (request bound function, parfois notée $G(t)$),

– la demande processeur des tâches devant se terminer avant la date t (l'échéance est avant ou à la date t), qui sera notée dbf (demand bound function, parfois notée $H(t)$).

La première fonction est particulièrement utile pour analyser les systèmes à priorité fixe, tandis que la seconde est utile pour analyser l'ordonnancement produit par EDF. Nous nous limitons aux tâches à échéance contrainte ($D_i \leq T_i$ - constrained deadline) et à départ différé (asynchronous) pour illustrer les définitions de ces deux fonctions.

La demande processeur des tâches réveillées dans l'intervalle de temps $[0, t[$ repose sur le nombre k d'activation d'une tâche τ_i dans cet intervalle de temps. Cela nécessite de connaître la dernière requête activée avant la date t. Les deux inégalités suivantes permettent de déterminer k :

$$r_i + (k-1)T_i < t \qquad \Rightarrow \qquad k < \frac{t - r_i}{T_i} + 1$$

$$r_i + kT_i \geq t \qquad \Rightarrow \qquad k \geq \frac{t - r_i}{T_i}$$

Nous devons de plus assurer $k \geq 0$. Ces inégalités seront respectées pour :
$k = \max(0, \lceil \frac{t-r_i}{T_i} \rceil)$. Notons que les requêtes comptabilisées ne sont pas nécessairement terminées à la date t. La durée maximum cumulée de travail processeur associée aux réveils de τ_i sur l'intervalle de temps $[0, t[$ est notée rbf (request bound function) :

$$rbf(\tau_i, t) = \max\left(0, \left\lceil \frac{t - r_i}{T_i} \right\rceil\right) C_i$$

La fonction de travail du processeur sur l'intervalle $[0, t[$ tient compte des requêtes de toutes les tâches :

$$W(t) = \sum_{j=1}^{n} rbf(\tau_j, t) \tag{4.1}$$

Le travail processeur est une fonction en escalier. La droite affine $f(t) = t$ définit la capacité maximum de traitement du processeur (avec une vitesse unitaire). Ainsi, lorsque $W(t) = t$, alors à cette date, le processeur a terminé toutes les requêtes réveillées dans l'intervalle $[0, t[$.

Pour les systèmes à priorité fixe, la fonction de travail $W_i(t)$ est la durée cumulée des tâches de priorité supérieure ou égale à i et réveillées dans l'intervalle $[0, t[$:

$$W_i(t) = \sum_{j=1}^{i} rbf(\tau_j, t) \tag{4.2}$$

La fonction $dbf(t_1, t_2)$ (demand bound function) est la durée cumulée des requêtes dont la date d'activation et l'échéance sont dans l'intervalle $[t_1, t_2]$. Nous pouvons définir le nombre k d'activation d'une tâche τ_i. Pour déterminer les requêtes dont les échéances surviennent dans un intervalle $[0, t]$, nous identifions la dernière requête de la tâche τ_i avec les deux inégalités suivantes :

$$\begin{aligned} r_i + (k-1)T_i + D_i \leq t \quad &\Rightarrow \quad k \leq \frac{t - r_i - D_i}{T_i} + 1 \\ r_i + kT_i + D_i > t \quad &\Rightarrow \quad k > \frac{t - r_i - D_i}{T_i} \end{aligned}$$

Nous savons aussi que $k \geq 0$. Ces inégalités seront respectées pour la tâche τ_i : $k = \max\left(0, \lfloor \frac{t - r_i - D_i}{T_i} \rfloor + 1\right)$. Nous avons donc la demande processeur des requêtes d'échéances inférieures ou égales à t dans l'intervalle $[0, t[$:

$$dbf(0, t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - r_i - D_i}{T_i} \right\rfloor + 1\right) \times C_i \tag{4.3}$$

Exemple 8 *Considérons un système de trois tâches à départ simultané dont les paramètres sont indiqués dans le tableau 4.4. La colonne R_i donne les temps de réponse des tâches lorsqu'elles sont ordonnancées avec des priorités fixes. Nous constatons que la tâche τ_3 n'est pas ordonnançable (alors qu'elle le serait avec une échéance égale à la période). La figure 4.4 représente les fonctions $W_3(t) = rbf(t)$ et $dbf(t)$.*

tâches	C_i	D_i	T_i	R_i
τ_1	2	10	10	2
τ_2	10	25	30	14
τ_3	55	100	120	119

Tableau 4.4. *Système de tâches. La colonne R_i indique les temps de réponse des tâches avec un ordonnanceur à priorité fixe*

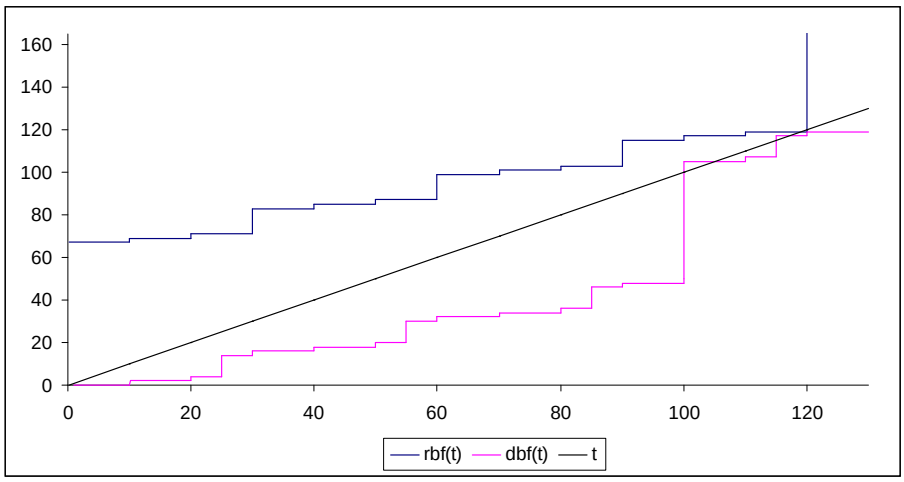

Figure 4.4. *Fonctions $rbf(t)$, $dbf(t)$ et $f(t) = t$ pour le système de tâches du tableau 4.4*

En raisonnant sur des inégalités simples comme nous l'avons fait dans les paragraphes précédents, nous pouvons généraliser les formules précédentes pour traiter des systèmes plus complexes, tels que ceux introduisant par exemple des paramètres supplémentaires dans la définition des tâches comme la gigue sur activation ou des temps de blocage associés à l'attente devant les ressources critiques (c.-à-d. le facteur de blocage). Il convient toutefois d'être vigilant afin d'assurer que les valeurs calculées par les fonctions rbf ou dbf sont des bornes supérieures de la demande processeur, sinon le test d'ordonnançabilité correspondant ne serait pas correct. Ceci nécessite de caractériser **le(s) pire(s) scénario(s)** d'arrivée des requêtes engendrant la plus grande activité du processeur.

4.4.2.3. *Tests approchés*

L'objectif d'un test approché est de fournir une **décision approchée** au problème d'ordonnançabilité : si le test approché renvoie ordonnançable, alors les tâches respecteront leurs contraintes temporelles, sinon on ne peut pas conclure. La principale motivation pour concevoir un test approché est de définir des algorithmes de test avec une complexité algorithmique plus faible, soit parce que le test exact nécessite un temps de calcul exponentiel ; soit parce que le test doit être utilisé en-ligne pour contrôler l'admission de nouvelles tâches périodiques.

Les tests approchés vont donc utiliser des valeurs approchées de la demande processeur (c.-à-d. des bornes supérieures des fonctions présentées dans le paragraphe précédent) et limiter le nombre d'itérations nécessaires pour prendre une décision. Ces deux opérations vont introduire du **pessimisme** dans les méthodes d'analyse.

Il est important (et peu rassurant) de constater qu'aucune estimation quantitative du pessimisme des tests approchés n'est en général rigoureusement proposée dans la littérature. Les évaluations reposent uniquement sur des simulations numériques qui ne comparent pas le test approché avec un test exact. De façon générale, les tests approchés d'ordonnançabilité conduisent à surdimensionner les systèmes afin de satisfaire la condition suffisante d'ordonnançabilité définie dans le test de validation !

Nous rappelons tout d'abord quelques définitions sur l'approximation polynomiale, puis nous illustrons les tests avec garantie de performance sur les tâches à priorité fixe.

Une estimation du pessimisme des méthodes approchées peut être obtenue en utilisant des **algorithmes d'approximation** (ou approchés). Ces algorithmes sont utilisés pour résoudre de façon approchée des problèmes d'optimisation. L'intérêt d'un algorithme d'approximation est de posséder une **garantie de performance** vis-à-vis d'une méthode exacte. Précisément, soit A un algorithme approché et OPT une méthode exacte, alors la borne d'erreur ϵ ($0 < \epsilon < 1$) de l'algorithme A pour toute configuration I du problème d'optimisation est définie par :

$$\frac{|A(I) - OPT(I)|}{OPT(I)} \leq \epsilon$$

Un algorithme approché a une garantie de performance bornée par le **ratio** suivant : $r_A = 1 + \epsilon$ (pour un problème de minimisation). Ce ratio définit donc les pires résultats que pourra atteindre l'algorithme approché A en considérant toutes les configurations possibles d'un problème d'optimisation. Une approximation polynomiale est un algorithme avec un ratio constant. Un schéma d'approximation est un algorithme paramétrique, de paramètre ϵ, qui peut s'approcher aussi près que possible de la valeur optimale de la fonction optimisée. Le ratio d'un schéma d'approximation polynomiale (PTAS - Polynomial Time Approximation Scheme) s'écrit sous la forme : $r_A \leq 1 + \epsilon$.

Un schéma d'approximation est complet (FPTAS - Fully Polynomial Time Approximation Scheme) s'il est un PTAS et que l'algorithme est en plus polynomial en fonction du paramètre $1/\epsilon$. Un FPTAS est le meilleur résultat d'approximation pouvant être obtenu pour résoudre un problème $\mathcal{NP}$-Difficile. Nous renvoyons à [GAR 79] pour des compléments sur la complexité et l'approximabilité des problèmes.

Depuis plusieurs années, les algorithmes d'approximation intéressent les concepteurs de tests afin de garantir les performances dans le pire cas des tests approchés. Toutefois, tester la faisabilité d'un système de tâches est un problème de décision, alors que les algorithmes approchés concernent les problèmes d'optimisation. Bien que le temps de réponse soit un critère quantitatif, il n'existe pas à notre connaissance de calcul de pire temps de réponse

approché avec une garantie constante de performance par rapport au pire temps de réponse exact. Cependant, des tests approchés reposant sur l'analyse de la demande processeur ont été proposés, comme nous le verrons plus loin.

4.5. Analyse du facteur d'utilisation

Dans ce paragraphe, les tests d'ordonnançabilité basés sur les facteurs d'utilisation sont traités. Ces tests concernent les algorithmes décrits dans le paragraphe 4.2. Ces tests sont exacts ou approchés.

4.5.1. *Tests exacts*

Le test d'ordonnançabilité pour les algorithmes EDF et LLF pour l'ordonnancement de configurations de tâches indépendantes, périodiques et à échéance sur requête est celui de LIU et LAYLAND en 1973 [LIU 73, DEV 00] (condition nécessaire et suffisante) :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{4.4}$$

Ainsi, pour les configurations de tâches à départ simultané et à échéance sur requête, le taux d'utilisation maximum que peut atteindre le processeur sous *Earliest Deadline First* est de 100%.

4.5.2. *Tests approchés*

Le test d'ordonnançabilité de LIU et LAYLAND en 1973 [LIU 73, DEV 00] établit une condition suffisante d'ordonnançabilité pour toute configuration de tâches périodiques, indépendantes, à échéance sur requête et à départ simultané (ce test est une condition suffisante mais pas nécessaire) :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{4.5}$$

Un autre test approché (donc condition uniquement suffisante) a été établi par BINI et *al.* [BIN 03] pour l'ordonnancement de configuration de tâches périodiques :

$$\prod_{i=1}^{n} (U_i + 1) \leq 2$$

Le test LIU et LAYLAND en 1973 [LIU 73] (Formule 4.5) a été adapté pour tester les configurations de tâches indépendantes et périodiques et à départ simultané. Ce test n'est qu'une condition suffisante :

$$U_\ell = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}} - 1)$$

Pour EDF, si les tâches sont périodiques et indépendantes mais que l'échéance de toutes les tâches est inférieure à sa période alors le test devient (condition suffisante) :

$$U_\ell = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \tag{4.6}$$

Si enfin, l'échéance de toutes les tâches est supérieure à la période, le test 4.4 reste une condition nécessaire et suffisante [BAR 90a].

Des extensions à ce test d'ordonnançabilité existent (partage de ressources,...), nous renvoyons à [DEV 03, LIU 00] pour des informations complémentaires.

4.6. Analyse du temps de réponse

4.6.1. *Tests exacts*

Le temps de réponse d'une requête est la différence entre sa date de fin et sa date d'activation. Le pire temps de réponse d'une tâche est le plus grand temps de réponse de ses requêtes. L'ordonnancement étant périodique, le nombre de valeurs différentes des temps de réponse des requêtes d'une tâche est fini et calculable.

4.6.1.1. *Ordonnancement à priorité fixe*

Comme nous l'avons vu précédemment, le calcul pratique du temps de réponse nécessite toujours de caractériser le(s) scénario(s) d'arrivée des tâches conduisant au pire temps de réponse de la tâche étudiée, déterminer une fonction d'analyse de la durée cumulée des tâches correspondant au(x) pire(s) scénario(s). Nous illustrons le calcul du pire temps de réponse d'une tâche τ_i dans un système à priorité fixe [JOS 86].

Lemme 1 *(Scénario) Le pire temps de réponse d'une tâche τ_i survient d'une période d'activité de niveau i débutant par un instant critique.*

Theorème 1 *(Test) Le pire temps de réponse de τ_i est défini par le plus petit point fixe de l'équation :*

$$W_i(t) = t$$

Où la fonction $W_i(t)$ est définie dans la formule 4.2.

En pratique pour chaque tâche τ_i, résoudre cette équation se fait par approximation successive en définissant la suite :

$$\begin{aligned} R_i^0 &= \sum_{j=1}^{i} C_j \\ R_i^{n+1} &= \sum_{j=1}^{i} rbf(\tau_j, R_i^n) \end{aligned} \tag{4.7}$$

L'algorithme calcule par récurrence les valeurs successives de R_i^n et stoppe la récurrence lorsque $R_i^n = R_i^{n+1}$. Alors, le pire temps de réponse de τ_i, noté R_i^* est égal à R_i^n. Il reste finalement à démontrer que $R_i^* \leq D_i$. La complexité algorithmique du calcul du temps de réponse d'une tâche est pseudo-polynomiale : $O(n \sum_{i=1}^{n} C_i)$ pour $U < 1$ [SPU 96a]. Puisque les pires temps de réponse des tâches sont calculés en séquence, la complexité du test d'ordonnançabilité est la même que celle pour analyser une tâche. Notons que l'existence d'un algorithme polynomial est un problème ouvert.

Exemple 9 *Sur le système de tâches représenté dans le tableau 4.4, les temps de réponse sont calculés en 3 itérations maximum et les temps de réponse correspondants sont indiqués dans la colonne R_i de ce même tableau.*

4.6.1.2. *Ordonnancement* EDF

Contrairement au système à priorité fixe, le pire temps de réponse d'une tâche ordonnancée par EDF ne survient pas nécessairement dans la première période d'activité du processeur [SPU 96a] (celle initiée par l'instant critique où toutes les tâches sont réveillées simultanément). Le pire temps de réponse d'une tâche peut être calculé en construisant (par simulation) l'ordonnancement EDF. Mais l'algorithme résultant sera exponentiel. À notre connaissance, aucun algorithme polynomial ou pseudo-polynomial n'est encore connu pour calculer le pire temps de réponse exact des tâches périodiques ordonnancées par EDF (le problème peut être résolu en temps pseudo-polynomial pour les tâches périodiques [SPU 96a].

4.6.2. *Tests approchés*

Un test d'ordonnançabilité fournit une réponse binaire : **ordonnançable** ou **non ordonnançable**. Par contre, le temps de réponse est un critère quantitatif. Celui-ci peut donc être utilisé pour définir le ratio d'approximation du calcul du pire temps de réponse d'une tâche. À notre connaissance, il n'existe pas de résultat dans la littérature de calcul d'un pire temps de réponse approché avec une garantie de performance par rapport au pire temps de réponse exact.
La prise en compte de dates d'activation, ressources,... conduisent en général à des tests approchés car les scénarios pire cas considérés ne surviennent pas nécessairement dans la vie du système.

4.7. Analyse de la demande processeur

4.7.1. *Tests exacts*

L'analyse de la demande processeur est un test d'ordonnançabilité consistant à vérifier que toutes les requêtes devant s'exécuter dans tout intervalle de temps ne dépassent pas la capacité du processeur (c.-à-d. la longueur de l'intervalle considéré). Contrairement à l'analyse du temps de réponse, l'analyse de la demande processeur analyse toutes les tâches simultanément.

4.7.1.1. *Ordonnancement à priorité fixe*

Nous allons présenter les principes de cette analyse à travers l'exemple des tâches à priorité fixe et à départ simultané [LEH 89].

Lemme 2 *(Scénario) Il est suffisant d'analyser l'intervalle de temps* $[0, D_i)$ *pour savoir si* τ_i *respectera ou non son échéance.*

Theorème 2 *(Test de Lehoczky, Sha et Ding) Dans un système de tâches à départ simultané, une tâche* τ_i *est ordonnançable si, et seulement si, il existe un instant* $t \in (0, D_i]$ *tel que* $W_i(t) \leq t$*.*

Cela revient donc à rechercher la valeur minimum de la fonction $W_i(t)/t$ dans l'intervalle $[0, D_i]$. La fonction de la demande processeur ne change de valeur qu'à des instant précis car l'équation 4.1 est une fonction en escalier. Le test va se limiter aux valeurs correspondant à des minima locaux de la fonction de la demande processeur. L'ensemble de ces valeurs définissent l'**ensemble des points d'ordonnancement** (Testing Set ou Scheduling Point Set) :

$$S_i = \{bT_j | j = 1 \ldots i, b = 1 \ldots \lfloor D_i/T_j \rfloor\} \cup \{D_i\} \tag{4.8}$$

Ainsi, vérifier que la tâche τ_i est ordonnançable nécessite de calculer :
$\min_{t \in S_i} \left(\frac{W_i(t)}{t} \right) \leq 1$. En conséquence, si une date $t \in S_i$ satisfait $W_i(t) \leq t$ alors τ_i est ordonnançable et il est inutile d'examiner d'autres points d'ordonnancement. Ainsi en pratique, seulement un sous-ensemble des points d'ordonnancement de S_i sera analysé. Mais, d'un point de vue complexité algorithmique, le nombre d'itérations du test est borné par le ratio : D_i/T_j. Donc, sa complexité algorithmique est pseudo-polynomiale. Le test complet se formule de la façon suivante :

$$\max_{i=1\ldots n} \left\{ \min_{t \in S_i} \left(\frac{W_i(t)}{t} \right) \right\} \leq 1$$

Exemple 10 *Nous illustrons ce test sur l'analyse de la tâche* τ_3 *du système de tâches représenté dans le tableau 4.4. Les ensembles de tests sont :*

$$\begin{aligned} S_1 &= \{10\} \\ S_2 &= \{10, 20, 25, 30\} \\ S_3 &= \{10, 20, 30, 40, 50, 60, 70, 80, 90, 100\} \end{aligned}$$

Les calculs de $W_3(t)/t$ *pour l'ensemble* S_3 *sont inscrits dans le tableau 4.5. Le système est non ordonnançable puisqu'aucune date* $t \in S_3$ *ne conduit à vérifier la condition :*
$W_3(t)/t \leq 1$.

t	10	20	30	40	50	60	70	80	90	100
$W_1(t)/t$	0,2									
$W_1(t)/t$	1,2	0,7	0,5							
$W_3(t)/t$	6,7	3,4	2,3	2,0	1,7	1,4	1,4	1,2	1,14	1,1

Tableau 4.5. *Exécution du test associé au théorème 2 pour analyser le système de tâches du tableau 4.4.*

Nous renvoyons à [MAN 98] pour une présentation des algorithmes et une variante de ce test (permettant d'avoir une complexité indépendante des valeurs des paramètres, mais exponentielle - $O(n2^n)$). On pourra aussi consulter [BIN 04] pour avoir des informations complémentaires.

4.7.1.2. *Ordonnancement sous* EDF

Les tâches étant à départ simultané, la plus grande charge processeur survient au démarrage de l'application. Précisément, la demande processeur dans le cas de tâches à départ simultané vérifie [BAR 90b] :

Lemme 3 *(Scénario)*

$$dbf(0, t_2 - t_1) \geq dbf(t_1, t_2) \qquad \forall t_1, t_2 \quad \textit{et} \quad t_1 \leq t_2$$

Nous pouvons simplifier l'expression du calcul de la demande du processeur puisque nous supposons $r_i = 0$ et $D_i \leq T_i$, $1 \leq i \leq n$:

$$dbf(0,t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i \tag{4.9}$$

$$= \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i \tag{4.10}$$

Ceci conduit à définir un intervalle d'étude pour EDF débutant à l'instant critique 0, jusqu'à $H = ppcm(T_i)$, puisque l'ordonnancement est périodique de période H. En relâchant la contrainte d'intégralité de l'équation 4.10, il est démontré dans [BAR 90b] que l'intervalle d'étude peut se limiter dans le cas $U < 1$ à l'intervalle $[0, t_{lim}]$, avec :

$$t_{lim} = \frac{U}{1 - U} \max_{i=1..n}(T_i - D_i)$$

Ceci permet d'établir le test d'ordonnançabilité suivant :

Theorème 3 *(Test de Baruah, Howell et Rosier) Un système de tâches périodiques et à départ simultané est ordonnançable avec un facteur d'utilisation $U < 1$ si, et seulement si :*

$$dbf(0,t) \leq t \qquad \forall t, 0 < t < t_{lim}$$

Exemple 11 *Nous illustrons le fonctionnement de ce test sur le système de tâches du tableau 4.4. Le facteur d'utilisation du processeur est $U = 0,99$ et ceci conduit à la valeur $t_{lim} = 1980$ (On remarque que le ppcm des périodes est égal à $H = 120$. Cette valeur peut être utilisée comme limite de temps puisque les tâches sont à départ simultané). Nous devons donc tester toutes les dates t entre 0 et H pour vérifier la condition $dbf(t) \leq t$. Si pour une valeur de t, cette inégalité n'est pas vérifiée alors le système est non ordonnançable. Par contre, si le test est positif pour toutes les dates t entre 0 et H alors le système est ordonnançable. Pour les dates $t \in [0, 100[$, toutes les inégalités sont respectées. Par contre à la date $t = 100$, nous avons : $dbf(0,t) = 105 > t$ (nous pouvons aussi directement le constater sur la figure 4.4 : la courbe $dbf(0,t)$ passe au-dessus de la droite représentant la capacité du processeur $f(t) = t$). Le système de tâches n'est donc pas ordonnançable sous EDF.*

Sous l'hypothèse U constant, la complexité de ce test est $O\left(n \max_{i=1..n}(T_i - D_i)\right)$, l'algorithme est pseudo-polynomial. Ce test a été amélioré dans [RIP 96, PAR 04], mais ces algorithmes sont eux aussi pseudo-polynomiaux. Notons que l'existence d'un algorithme fortement polynomial est un problème ouvert.

4.7.1.3. *Généralisation*

Les tests exacts présentés reposent sur l'analyse de la demande processeur, exploitent des propriétés permettant de limiter l'intervalle d'étude et enfin la localisent dans la première période d'activité du processeur (cas de tâches à départ simultané). En l'absence de caractérisation précise de la période d'étude, l'analyse de la demande processeur doit être généralisée afin d'établir un test d'ordonnançabilité. Ceci passe notamment par la définition plus générale de la fonction de la demande cumulée du processeur.

Définition 18 *Fonction de demande du processeur (Demand Bound Function). Soit τ_i une tâche, la fonction de demande du processeur $dbf(t_1, t_2)$ est la durée maximum cumulée d'exécution des tâches qui ont leurs réveils et échéances dans un intervalle de durée $[t_1, t_2]$.*

Cette définition de la fonction $dbf(t_1, t_2)$ permet de définir un test simple pour les tâches indépendantes, de façon analogue au paragraphe précédent. Un test général peut être formulé de la façon suivante :

Theorème 4 *Un système de tâches est ordonnançable si, et seulement si :*

$$\forall t_1 < t_2 \qquad dbf(t_1, t_2) \leq t_2 - t_1 \tag{4.11}$$

Remarquons que dans le cas général, pour montrer que le système est non ordonnançable il est suffisant de trouver une valeur de t telle que l'inégalité 4.11 ne soit pas satisfaite. La généralisation à des tâches dépendantes avec des structures conditionnelles a été faite dans [BAR 03].
Toutefois, toute généralisation soulève deux questions :

– Comment calculer efficacement la fonction dbf ?

– Comment choisir un ensemble de points d'ordonnancement aussi petit que possible et qui soit suffisant pour garantir la correction du test défini par l'équation 4.11 ?

4.7.2. *Tests approchés (Approximation polynomiale)*

Un test d'ordonnançabilité fondé sur l'analyse de la demande processeur n'est pas un problème d'optimisation, mais un problème de décision (c.-à-d. qui retourne une valeur binaire). Toutefois, les techniques d'approximation vont pouvoir être utilisées, moyennant une adaptation de la définition de la garantie de performance. Nous présentons dans la suite le schéma d'approximation polynomiale de [ALB 04, FIS 05] qui utilise le paramètre ϵ avec la sémantique suivante : si le test répond **ordonnançable** alors le système est ordonnançable quel que soit son comportement à l'exécution. Et si le test répond **non ordonnançable**, alors il est non ordonnançable avec certitude sur un processeur plus lent (avec la vitesse $1-\epsilon$). Mais sur un processeur de vitesse unitaire, aucune décision ne peut être prise.
Nous illustrons cette approche sur l'ordonnancement à priorité fixe [FIS 05]. La fonction $rbf(\tau_i, t)$ est une fonction en escalier non décroissante. Le nombre de paliers dans cette fonction n'est pas borné polynomialement dans la taille du système à ordonnancer. Un moyen simple de définir un schéma d'approximation polynomiale est de ne considérer qu'un nombre borné k de paliers. Au-delà, une fonction linéaire (continue) sera utilisée pour définir une borne supérieure de $rbf(\tau_i, t)$. Le nombre de paliers va être défini à l'aide du paramètre d'erreur ϵ :

$$k = \left\lceil \frac{1}{\epsilon} \right\rceil + 1$$

Nous pouvons maintenant définir l'approximation de la demande cumulée de la demande processeur de la tâche τ_i, qui sera notée $\overline{rbf}(\tau_i, t)$:

$$\overline{rbf}(\tau_i, t) = rbf(\tau_i t) \quad \text{Si} \quad t \leq (k-1)T_i \tag{4.12}$$

$$= C_i + t\frac{C_i}{T_i} \quad \text{Sinon} \tag{4.13}$$

La demande processeur approchée, est alors définie par :

$$\overline{W}_i(t) = C_i + \sum_{j=1}^{i-1} \overline{rbf}(\tau_i, t) \tag{4.14}$$

ϵ	0,01	0,1	0,2	0,3	0,4	0,5
k	101	11	6	5	4	3

Tableau 4.6. *Nombre de paliers de la fonction $rbf(\tau_i, t)$ considérée avant la linéarisation de la fonction dans le test approché de Fisher et Baruah*

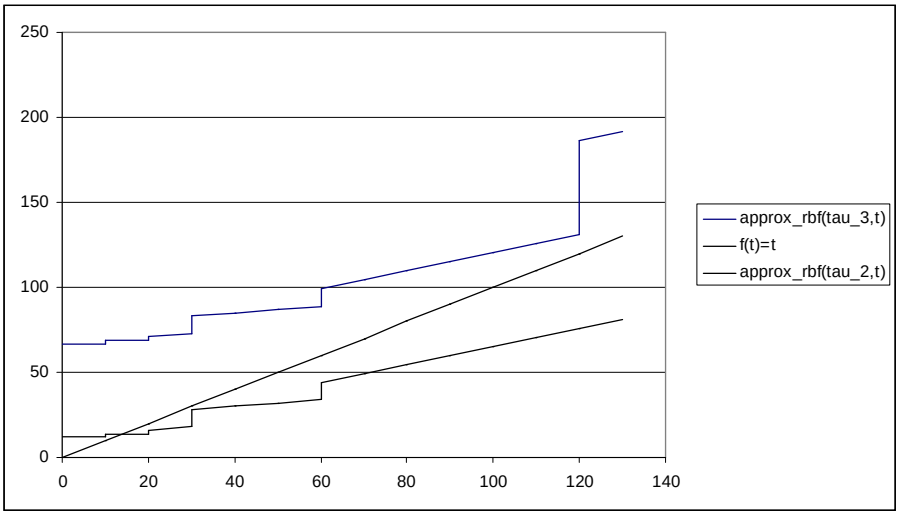

Figure 4.5. *Approximation de la demande processeur dans le test approché de* FISHER *et* BARUAH. *$\epsilon = 0,5$ ($k = 3$), le test conduit à l'ordonnançabilité de τ_2, mais τ_3 n'est pas ordonnançable sur un processeur de vitesse $(1 - \epsilon)$.*

Pour terminer le test, Fisher et Baruah utilisent l'analyse de la demande processeur avec un ensemble de points d'ordonnancement (Testing Set) possédant un nombre polynomial d'entrées dans la taille du système de tâches à analyser et du paramètre de précision $1/\epsilon$:

$$\overline{S}_i = \{bT_j | j = 1 \dots i-1, b = 1 \dots k\} \tag{4.15}$$

Exemple 12 *Le tableau 4.6 donne la valeur de k en fonction de la borne d'erreur ϵ. Nous donnons les figures 4.5 et 4.6, le graphique des fonctions $\overline{W}_2(t)$ et $\overline{W}_3(t)$ pour le système de tâches présenté dans le tableau 4.4 et des valeurs d'epsilon égales à 0,5 et 0,3.*

L'algorithme de test s'implémente très facilement en $O(n^2/\epsilon)$. Clairement, si ϵ est proche de 0, alors le nombre d'itérations effectuées par l'algorithme est très grand, mais le nombre d'itérations est polynomial en $1/\epsilon$. En conséquence, cet algorithme paramétrique est un FP-TAS. Le choix du paramètre de précision ϵ est donc primordial pour obtenir un test rapide et avec une bonne garantie de performance.

Le cas des tâches avec des échéances arbitraires (c.-à-d. telles que l'échéance D_i et la période T_i ne sont pas reliées par une contrainte) est présenté dans [FIS 05]. Par souci de concision, nous renvoyons à [ALB 04] pour l'approximation de la demande processeur pour EDF.

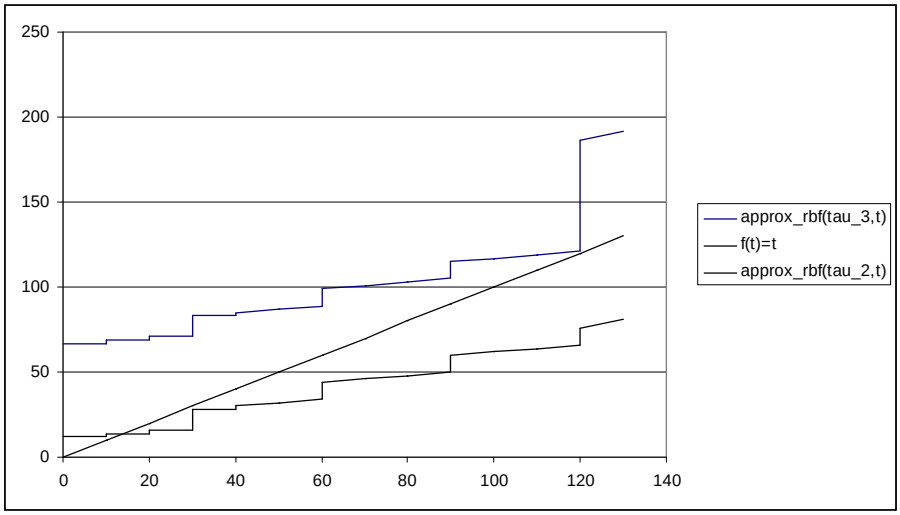

Figure 4.6. *Approximation de la demande processeur dans le test approché de* FISHER *et* BARUAH. $\epsilon = 0,3$ $(k = 5)$*, le test conduit à l'ordonnançabilité de* τ_2*, mais* τ_3 *n'est pas ordonnançable sur un processeur de vitesse* $(1 - \epsilon)$.

4.8. Complexité des problèmes d'ordonnançabilité

Nous pensons que l'établissement d'un test d'ordonnançabilité devient difficile à concevoir lorsque l'algorithme d'ordonnancement n'est pas *robuste* (c.-à-d. sujet aux anomalies d'ordonnancement *cf.* paragraphe 4.3.4) pour le problème considéré et que le problème d'ordonnançabilité est *difficile*. À notre connaissance, ces deux critères ne sont pas connus comme étant dépendants. Dans les deux cas, le calcul de la demande processeur (fonctions rbf ou dbf) doit donc reposer sur un scénario conduisant à la pire demande processeur. Un résultat analytique doit impérativement être construit pour garantir la correction du test.

Déterminer la complexité du problème d'ordonnançabilité permet d'identifier quel type d'algorithme doit être construit pour analyser un système de tâches. Si le problème est $\mathcal{NP}$-difficile alors les tests seront généralement des tests approchés car établir un test exact sera trop coûteux en temps de calcul et le(s) pire(s) scénario(s) sera(ont) difficile(s) à caractériser. Le paysage est paradoxalement compliqué en ordonnancement de tâches périodiques puisque les problèmes sont soit dans $\mathcal{NP}$, soit dans co-$\mathcal{NP}$, ou bien non connus comme étant dans l'une de ces deux classes de problèmes.

Exemple 13 *Nous donnons une caractérisation simple de ces deux classes de problèmes vis-à-vis de l'ordonnançabilité des tâches à départ simultané et à échéance contrainte (les tests exacts ont été développés plus haut) :*

*– pour les systèmes ordonnancés avec des priorités fixes, il est possible de décider en temps polynomial qu'*une tâche est ordonnançable *(par un algorithme non déterministe). Le test de* LEHOCZKY, SHA *et* DING *(voir [LEH 89]) permet de résoudre ce problème en temps pseudo-polynomial. Supposons qu'un algorithme non déterministe (c.-à-d. l'oracle d'une machine de Turing non déterministe) nous donne un point d'ordonnancement* t *et une tâche* τ_i*, alors la tâche est ordonnançable si la condition* $W_i(t) \leq t$ *est vraie. Cette vérification*

s'effectue en temps polynomial, montrant ainsi que le problème d'ordonnançabilité est dans $\mathcal{NP}$.

– on sait, pour les systèmes ordonnancés par EDF *décidés en temps polynomial, qu'*une tâche n'est pas ordonnançable *(par un algorithme non déterministe). Supposons qu'un algorithme non déterministe nous donne un point d'ordonnancement t, alors la condition $dbf(0,t) > t$ permet de conclure que le système est non ordonnançable en temps polynomial [BAR 90b]. Ceci établit que le problème d'ordonnançabilité est dans co-$\mathcal{NP}$.*

Il est assez surprenant que l'analyse d'ordonnançabilité pour un système de tâches donné est soit dans $\mathcal{NP}$, soit dans co-$\mathcal{NP}$, en fonction de l'algorithme d'ordonnancement considéré (à priorité fixe ou EDF). Après tout, cela laisse un peu d'espoir sur l'existence d'un test polynomial ! Les deux exemples considérés précédemment sont ouverts du point de vue de leur complexité (ils ne sont pas connus $\mathcal{NP}$-difficiles et aucun algorithme polynomial n'est connu).
Notons par ailleurs que l'ordonnançabilité des tâches en mode non préemptif est $\mathcal{NP}$-Difficile au sens fort et que l'ordonnancement de tâches à départ différé est co-$\mathcal{NP}$-Complet au sens fort (y compris en préemptif), et enfin l'ordonnancement de tâches avec suspension est $\mathcal{NP}$-Difficile au sens fort [RIC 03, RID 04].

4.9. Conclusion

Les travaux sur l'ordonnancement monoprocesseur sont multiples et ainsi de nombreux résultats sont connus. Ces résultats peuvent être utilisés pour valider une multitude de systèmes industriels. Mais, dans beaucoup de ces pratiques, leurs utilisations nécessitent souvent d'étendre le modèle des tâches pour tenir compte des contraintes de l'application. Ainsi de nouvelles études restent à mener sur les systèmes monoprocesseurs afin d'intégrer efficacement des facteurs pratiques [COT 00, LIU 00, SHA 04].

4.10. Bibliographie

[ABE 98] ABENI L., « Server Mechanisms for Multimedia Applications », *Technical Report RETIS TR98-0101, Scuola Superiore S.Anna, Pisa, Italy*, 1998.

[ALB 04] ALBERS K., SLOMKA F., « An event stream driven approximation for the analysis of real-time systems », *Euromicro Int. Conf. on Real-Time Systems (ECRTS'04)*, 2004.

[AND 03] ANDERSSON B., Static-priority scheduling on multiprocessors, PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 2003.

[AUD 93] AUDSLEY N., BURNS A., RICHARDSON M., TINDELL K., WELLINGS A., « Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling », *Software Engineering Journal*, vol. 8, n°5, p. 284-292, 1993.

[BAK 91] BAKER T., « Stack-based scheduling of real-time processes », *Journal of Real-Time Systems*, vol. 3, n°1, p. 67-99, mars 1991.

[BAR 90a] BARUAH S., MOK A., ROSIER L., « Preemptively scheduling hard-real-time sporadic tasks on one processor », *In Proceedings of the* 11^{th} *IEEE Real-Time Systems Symposium (RTSS'90)*, p. 182-190, 1990.

[BAR 90b] BARUAH S., ROSIER L., HOWELL R., « Algorithms and Complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor », *Journal of Real-Time Systems*, vol. 1, p. 301-324, 1990.

[BAR 03] BARUAH S., « Dynamic- and static-priority scheduling of recurring real-time tasks », *Journal of Real-Time Systems*, vol. 24, n°1, p. 93-128, 2003.

[BAR 04] BARUAH S., GOOSSENS J., « Scheduling Real-time Tasks : Algorithms and Complexity », *In Handbook of Scheduling : Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (edt). Chapman Hall/CRC Press*, 2004.

[BIN 03] BINI E., BUTTAZZO G., BUTTAZZO G., « Rate Monotonic Analysis : The Hyperbolic Bound », *IEEE Transactions On Computers*, vol. 52, n°7, page , juillet 2003.

[BIN 04] BINI E., BUTTAZZO G., « Schedulability Analysis of Periodic Fixed-Priority Systems », *IEEE Transactions on Computers*, vol. 53, n°11, page , novembre 2004.

[BRI 06] BRIL R., « Existing Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Refuted », *Work-In-Progress Session of the 18th Euromicro Conference on Real-Time Systems*, vol. 1, n°5–7, p. 1–4, Juillet 2006.

[BUR 95] BURNS A., « Preemptive Priority-Based Scheduling : An Appropriate Engineering Approach », *in Advances in Real-Time Systems, S.H. Son, Ed., Prentice Hall, New Jersey*, p. 225-248, 1995.

[CHA 05] CHAN W., LAM T., LIU K., WONG W., « New Resource Augmentation Analysis of the Total Stretch of SRPT and SJF in Multiprocessor Scheduling », *30th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, page , 2005.

[CHE 89] CHETTO H., CHETTO M., « Some results of the earliest deadline scheduling algorithm », *IEEE Transitions on Software Engineering*, vol. 15, n°10, p. 1261–1269, 1989.

[CHE 90] CHETTO H., SILLY M., BOUCHENTOUF T., « Dynamic scheduling of real-time task under precedence constraints », *Real Time Systems*, vol. 2, n°2, p. 181-194, 1990.

[COT 00] COTTET F., DELACROIX J., KAISER C., MAMMERI Z., *Ordonnancement temps réel*, Hermès Editions, 2000.

[DER 74] DERTOUZOS M., « Control robotics : the procedural control of physical processors », *IFIP Congress*, p. 807-813, 1974.

[DER 89] DERTOUZOS M., MOK A., « Multiprocessor on-line scheduling of hard real-time tasks », *IEEE Trasnactions on Software Engineering*, vol. 15, p. 1497-1506, 1989.

[DEV 00] DEVILLERS R., GOOSSENS J., « Liu and Layland's schedulability test revisited », *Information Processing Letters*, vol. 73, n°5-6, p. 157-161, March 2000.

[DEV 03] DEVI U., « An Improved Schedulability Test for Uniprocessor Periodic Task Systems », *IEEE Euromicro Conf. on Real-Time Systems (ECRTS'03)*, p. 23-32, 2003.

[FIS 05] FISHER N., BARUAH S., « A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines », *proc. Proceedings of the EuroMicro Conference on Real-Time Systems, (ECRTS'05)*, 2005.

[GAR 79] GAREY M., JOHNSON D., *Computers and Intractability : a guide to the theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[GRA 69] GRAHAM R., « Bounds on multiprocessing timing anomalies », *SIAM Journal of Allied Mathematics*, vol. 17, n°2, p. 416-429, décembre 1969.

[JAC 55] JACKSON J., « Scheduling a production line to minimize maximum tardiness », *Management Science Research Project 43, UCLA*, janvier 1955.

[JEF 92] JEFFAY K., « Scheduling sporadic tasks with shared resources in hard-real-time systems », *In Proceeding of the* 13^{th} *IEEE Real-Time Systems Symposium*, p. 89-99, 1992.

[JOS 86] JOSEPH M., PANDYA P., « Finding response times in real-time systems », *Comp. Journal*, vol. 29, n°5, page , 1986.

[LAB 74] LABETOULLE J., « Un algorithme optimal pour la gestion des processus en temps réel », *Revue Française d'Automatique, Informatique et Recherche Opérationnelle*, p. 11-17, janvier 1974.

[LEH 87] LEHOCZKY J., SHA L., STROSNIDER J., « Enhanced Aperiodic Responsiveness in Hard Real-Time Environments », *Proceedings 8th IEEE Real-Time System Symposium, San Jose, California*, p. 261-270, décembre 1987.

[LEH 89] LEHOCZKY J., SHA L., DING Y., « The Rate Monotonic Scheduling Algorithm : Exact Characterization and Average Case behavior », *In Proceedings of the* 10^{th} *IEEE Real-Time Systems Symposium (RTSS'89)*, p. 166-171, 1989.

[LEU 80] LEUNG J., MERRILL M., « A note on preemptive scheduling of periodic, real-time tasks », *Information processing letters*, vol. 11, n°3, p. 115-118, 1980.

[LEU 82] LEUNG J., WHITEHEAD J., « On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks », *perf. Eval. (Netherlands)*, vol. 2, p. 237-250, 1982.

[LIU 73] LIU C., LAYLAND J., « Scheduling algorithms for multiprogramming in a hard real-time environment », *Journal of the ACM*, vol. 20, n°1, p. 40-61, 1973.

[LIU 00] LIU J., *Real-Time Systems*, Prentice hall, 2000.

[MAN 98] MANABE Y., AOYAGI S., « A feasibility decision algorithm for Rate Monotonic and Deadline Monotonic scheduling », *Journal of Real-Time Systems*, vol. 14, n°2, p. 171-181, 1998.

[MOK 83] MOK A., *Fundamental design problems of distributed systems for the hard real-time environment*, PhD Thesis, Massachusetts Institute of Technology, 1983.

[NAV 06] NAVET N., Ed., *Systèmes temps réel — Ordonnancement, réseaux et qualité de service*, vol. 2, Hermès, 2006, Chapitre 2 : Ordonnancement temps réel multiprocesseur (24 pages). ISBN 2–7462–1304–4.

[PAR 04] PARK M., CHO Y., « Feasibility analysis of hard real-time periodic tasks », *Journal of Systems and Softwares*, vol. 73, p. 89-100, 2004.

[PHI 97] PHILIPS C., STEIN C., TORNG E., WEIN J., « Optimal time-critical scheduling via resource augmentation », *prc. 29^{th} Ann. ACM Symp. on Theory of Computing*, p. 110–149, 1997.

[RAJ 89] RAJKUMAR R., « Task synchronization in real-time systems », *PhD Thesis, Carnegie Mellon University,*, août 1989.

[RAJ 91] RAJKUMAR R., « Synchronisation in Real-Time Systems : A priority Inheritance Approach », *Kluwer Academic Publishers*, 1991.

[RIC 01] RICHARD P., COTTET F., RICHARD M., « On-line scheduling of Real-Time Distributed Computers With Complex Communication Constraints », *ICECCS'2001, edited by Press, IEEE Computer, Skövde (Sweden)*, p. 26-34, 2001.

[RIC 02] RICHARD M., RICHARD P., GROLLEAU E., COTTET F., « Contraintes de précédences et ordonnancement monoprocesseur », *proc. Embedded Systems (RTS'02), Paris,* , n°1, p. 121-138, 2002.

[RIC 03] RICHARD P., « On the complexity of scheduling real-time tasks with self-suspensions on one processor », *IEEE Euromicro Conf. on Real-Time Systems (ECRTS'03)*, page 8p, 2003.

[RID 04] RIDOUARD F., RICHARD P., COTTET F., « Negative results for scheduling independent hard real-time tasks with self-suspensions », *25^{th} IEEE International Real-Time Systems Symposium (RTSS'04)*, 2004.

[RIP 96] RIPOLL I., CRESPO A., MOK A., « Improvement in feasibility testing for real-time tasks », *Journal of Real-Time Systems*, vol. 11, n°1, p. 19-39, 1996.

[SHA 88] SHA L., GOODENOUGH J., RALYA T., « An analytic approach to real-time software engineering », *Softw. Engin. Inst. Draft Report*, 1988.

[SHA 90] SHA L., RAJKUMAR R., LEHOCZKY J., « Priority inheritance protocols : An approach to real-time system synchronization », *IEEE Transactions on Computers*, vol. 39, n°9, p. 1175-1189, 1990.

[SHA 04] SHA L., ABDELZAHER T., ARZÈN K., CERVIN A., BAKER T., BURNS A., BUTTAZZO G., LEHOCZKY J., MOK A., « Real-Time Scheduling Theory : A Historical Perspective », *Journal of Real-Time Systems*, vol. 28, n°2, p. 101-156, décembre 2004.

[SPR 88] SPRUNT B., LEHOCZKY J., SHA L., « Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm », *Proceedings 9th IEEE Real-Time System Symposium, Hunstville*, p. 251-258, décembre 1988.

[SPR 89] SPRUNT B., SHA L., LEHOCZKY J., « Aperiodic Task Scheduling for Hard Real-Time Systems », *The Journal of Real-Time Systems*, p. 27-69, 1989.

[SPU 93] SPURI M., STANKOVIC J., « How to integrate precedence constraints and shared resources in real-time scheduling », *Technical Report UM-CS-1993-019, U. Mass*, 1993.

[SPU 94] SPURI M., BUTTAZZO G., « Efficient Aperiodic Service under Earliest Deadline Scheduling », *In Proceedings of the 15^{th} IEEE Real-Time Systems Symposium (RTSS'94), San Juan, Puerto Rico*, p. 2-11, décembre 1994.

[SPU 95] SPURI M., BUTTAZZO G., SENSINI F., « Robust Aperiodic Scheduling under Dynamic Priority Systems », *In Proceedings of the 16^{th} IEEE Real-Time Systems Symposium (RTSS'95), Pisa, Italy,*, p. 210-219, décembre 1995.

[SPU 96a] SPURI M., « Analysis of deadline scheduled real-time systems », *INRIA Reseach Report 2772*, page34p, 1996.

[SPU 96b] SPURI M., BUTTAZZO G., « Scheduling Aperiodic Tasks in Dynamic Priority Systems », *Journal of Real-Time Systems*, vol. 10, n°2, p. 179-210, mars 1996.

[STA 95] STANKOVIC J., SPURI M., DINATALE M., BUTTAZZO G., « Implications of classical scheduling results for real-time systems », *IEEE Computer*, vol. 28, n°6, p. 16-25, 1995.

[STR 95] STROSNIDER J., LEHOCZKY J., SHA L., « Algorithms and Complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor », *IEEE Transactions on Computers*, vol. 44, n°I, janvier 1995.

[TIN 92] TINDELL K., « An extendible approach for analyzing fixed priority hard real-time tasks », *Technical Report YCS189*, 1992.

Chapitre 5

Ordonnancement des tâches à suspension : Etat de l'art

5.1. Introduction

Les systèmes temps réel recèlent de tâches, qui au cours de leur exécution, lancent des opérations sur des processeurs externes. Pendant l'exécution de ces opérations, la tâche ne peut plus s'exécuter. Elle doit attendre la fin de ces opérations pour finir sa propre exécution. Ces tâches sont appelées *Tâches à suspension*.

Les études s'intéressant aux tâches à suspension sont peu nombreuses. De plus, elles se limitent bien souvent à adapter des tests d'ordonnançabilité établis pour l'ordonnancement de tâches périodiques déjà existantes, au problème des tâches à suspension.

Notre modèle, pour l'étude des tâches à suspension, reprend celui des tâches périodiques (vu dans le chapitre 4) et l'étend de la façon suivante :

– I dénote une configuration à n tâches à suspension (où n désigne un entier strictement positif).

– Les tâches sont considérées indépendantes.

– Chaque tâche τ_i ($1 \leq i \leq n$) arrive dans le système à la date r_i, et doit terminer son exécution avant la date limite de fin d'exécution notée $d_i = r_i + D_i$, où D_i est l'échéance relative à sa date d'arrivée.

– Enfin, chaque activation des requêtes de τ_i est séparée par une période T_i.

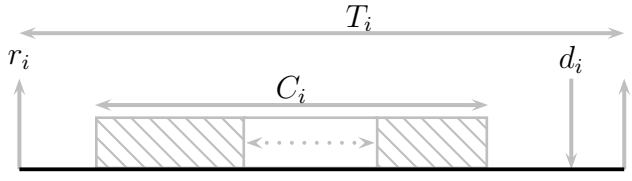

Figure 5.1. *Caractéristiques des tâches à suspension sous forme d'attente active*

Un moyen simple, mais réducteur de représenter les tâches à suspension, consiste à intégrer les durées de suspension à la durée d'exécution C_i (c.-à-d. que la tâche réalise une attente active). Ainsi, le modèle de représentation des tâches utilisé dans l'ensemble de ces études est bien souvent réducteur car limité à un seul bloc d'exécution noté C_i comme le montre la figure 5.1. Ce qui implique que pour les tâches qui effectuent des opérations externes, le temps nécessaire à la tâche en vue d'effectuer cette opération est compris à l'intérieur même du bloc d'exécution. Mais pendant qu'elle effectue une opération externe, une tâche n'utilise pas le processeur et par conséquent, elle peut laisser une autre tâche s'exécuter. Cette simplification de tâches n'est pas réaliste.

De plus, il est possible d'imaginer pour chaque tâche plusieurs suspensions pendant la même exécution (*cf.* figure 5.2).

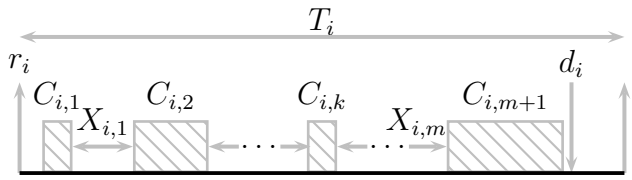

Figure 5.2. *Caractéristiques des tâches à suspension*

Cependant, dans la suite de cette étude, et pour simplifier le problème d'ordonnancement, nous considérons que les tâches ne peuvent se suspendre au plus qu'une seule fois. Ce modèle est représenté par la figure 5.3. Chaque tâche τ_i ($1 \leq i \leq n$) est composée de deux blocs d'exécution, appelés aussi sous-tâches (de longueur $C_{i,k}$, $1 \leq k \leq 2$), séparés par une suspension de durée maximale X_i pendant laquelle la tâche ne s'exécute pas. La valeur X_i est une borne supérieure et non pas la valeur exacte, car l'exécution de l'opération externe se fait sur un autre processeur. Par conséquent, elle entre en concurrence avec d'autres tâches qui veulent s'exécuter sur ce processeur et donc il n'y a pas de temps de réponse exacte, juste une borne supérieure du temps de réponse. La somme des longueurs des deux blocs d'exécution $C_{i,k}$ ($1 \leq k \leq 2$) est notée C_i. À l'intérieur de chaque bloc d'exécution, par définition il n'y a aucune suspension.

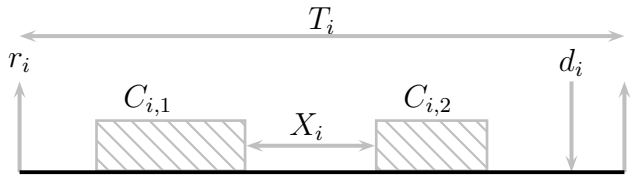

Figure 5.3. *Le modèle des tâches avec au plus une suspension*

Du problème explicité ci-dessus deux sous-problèmes peuvent se déduire et peuvent faire l'objet d'études plus approfondies :

1) Pour chaque tâche τ_i (avec $1 \leq i \leq n$), la durée $C_{i,1}$ est égale à zéro. Ce problème est également plus connu dans la littérature sous l'appellation du problème de gigue sur

activation (*release jitter*). Nous pouvons interpréter ce problème comme une nécessité pour chaque tâche τ_i d'attendre pendant une durée X_i avant de commencer l'exécution de la tâche. X_i (notée aussi J_i dans ce cas) est aussi appelée gigue sur activation. Dans ce cas particulier, la tâche τ_i arrive dans le système à la date r_i et n'est prête à s'exécuter qu'à la date $r_i + J_i$ et le temps processeur disponible pour l'exécution est $D_i - J_i$.

2) Pour chaque tâche τ_i (avec $1 \leq i \leq n$), la durée $C_{i,2}$ est égale à zéro. Ce problème est plus connu dans la littérature sous le nom de *delivery time* en ordonnancement classique. La tâche avant de finir son exécution doit attendre pendant une durée X_i. Dans ce cas particulier, la tâche τ_i arrive dans le système à la date r_i et doit avoir terminé son exécution avant la date $d_i - X_i$ pour lui permettre de se suspendre pendant X_i unités de temps avant sa date limite de fin d'exécution, d_i.

En ordonnancement en-ligne, comme nous l'avons vu précédemment, nous n'avons pas la connaissance de l'ensemble du problème. Ainsi, à un instant t donné, l'algorithme d'ordonnancement en-ligne ne connaît que l'ensemble des tâches disponibles, prêtes à être exécutées et celles qui n'ont pas encore achevé leur exécution. De plus, la durée de suspension d'une tâche n'est connue que lorsque celle-ci est terminée. Seule une borne supérieure X_i de la suspension est connue à l'activation de la tâche.

En hors-ligne, nous connaissons toutes les données du problème : l'ensemble de toutes les tâches ainsi que leurs caractéristiques dont les durées de suspension. L'ordonnancement est construit en supposant que les paramètres C_i et X_i sont fixes. Si, à l'exécution d'une tâche, une suspension dure moins longtemps (que la durée X_i), alors un temps creux est inséré dans l'ordonnancement pour que la durée de suspension X_i soit toujours vérifiée. Ce problème en hors-ligne se rapproche du problème des *time lags*. Les *time lags* sont des décalages temporels entre les requêtes étudiés entre autres dans [BRU 01] en ordonnancement classique.

La théorie de l'ordonnancement temps réel se focalise sur les tests d'ordonnançabilité pour s'assurer que toutes les échéances sont respectées. Plusieurs tests d'ordonnançabilité sont connus pour l'analyse de tâches à suspension. Des tests pour les algorithmes classiques existent déjà. Des tests d'ordonnançabilité concernant les algorithmes à priorité fixe sont basés sur le calcul du pire temps de réponse des tâches [KIM 95, LIU 00, PAL 98]. Une telle approche a aussi été développée au sujet de l'algorithme d'ordonnancement EDF [PAL 03]. Dans [DEV 03] un autre test est présenté, basé sur le facteur d'utilisation du processeur. Ce test se base sur les travaux de LIU [LIU 00] sur les algorithmes à priorité fixe et les adapte à *EDF*.

Dans le prochain paragraphe, nous dissocions pour plus de lisibilité, les tests d'ordonnançabilité basés sur les algorithmes à priorité fixe aux tests basés sur l'algorithme EDF. Pour chacun de ces tests d'ordonnançabilité établis dans la littérature, notre approche sera de définir dans un premier temps le modèle de tâche utilisé ainsi que le raisonnement utilisé pour son établissement. Dans un deuxième temps, nous présentons le test lui-même.

5.2. Tests d'ordonnançabilité

5.2.1. *Introduction*

L'un des principaux problèmes connus en temps réel est celui de l'ordonnançabilité. C'est-à-dire, pouvoir déterminer si une configuration de tâches est ordonnançable par un algorithme donné ou non. Plusieurs techniques ont été proposées et consistent à déterminer pour chaque algorithme (ou classe d'algorithmes) un test d'ordonnançabilité. Afin d'obtenir des tests précis, bien souvent, la complexité du test croît considérablement, ce qui le rend inutilisable. Ainsi, les tests d'ordonnançabilité ne sont que rarement des tests exacts, ce sont des tests approchés des tests exacts mais avec une complexité plus faible, les rendant utilisables. Par conséquent, si le test répond *oui*, la configuration de tâches testée est ordonnançable par l'algorithme (ou la classe d'algorithmes) étudié(e). S'il répond *non*, aucune conclusion ne peut être tirée quant à l'ordonnançabilité du système.

Quant à la présentation de chacun des tests d'ordonnançabilité, nous utilisons une configuration de tâches. En effet, pour chacun des tests, nous présentons son procédé de calculs au travers d'une même configuration. Les tâches de la configuration utilisée sont des tâches à départ simultané et à échéance sur requête. Soit I la configuration de tâches à suspension suivante :

$$\tau_1 : C_{1,1} = 1, X_1 = 1, C_{1,2} = 1, D_1 = T_1 = 8$$
$$\tau_2 : C_{2,1} = 3, X_2 = 3, C_{2,2} = 1, D_2 = T_2 = 40$$
$$\tau_3 : C_{3,1} = 1, X_3 = 2, C_{3,2} = 2, D_3 = T_3 = 80$$

Ces tests sont des tests d'ordonnançabilité basés sur le calcul du pire temps de réponse. Nous rappelons que le principe de ce test consiste d'une part à calculer pour chaque tâche le pire temps de réponse en utilisant la formule du test, puis, d'autre part, à vérifier que ce pire temps de réponse est inférieur à son échéance relative.

Les tests développés dans ce paragraphe calculent une borne supérieure la plus fine possible du pire temps de réponse de chaque tâche à suspension. Nous présentons les tests suivants :

– WELLINGS [AUD 93] : ce test n'est pas établi pour le problème des tâches à suspension, mais pour celui des gigues sur activation et il constitue la base des extensions aux tâches avec suspension ;

– MING *et al.* [MIN 94] : ce test considère la durée de suspension d'une tâche incluse dans sa durée d'exécution ;

– KIM et al. [KIM 95] : pour établir des tests d'ordonnançabilité, ils utilisent les travaux d'une part de WELLINGS [AUD 93] et d'autre part de MING *et al.* [MIN 94] ;

– JANE W. S. LIU [LIU 00] : ce test d'ordonnançabilité détermine pour chaque tâche, un temps de blocage dû à la suspension de la tâche elle-même et à celle des tâches plus prioritaires.

5.2.2. *Les travaux de* WELLINGS

WELLINGS *et al.* [AUD 93] étudient le problème d'ordonnancement de gigue sur activation (*cf.* paragraphe 4.3.1 et [BUR 95]). Ainsi, pour chaque tâche τ_i, le bloc d'exécution $C_{i,1} = 0$. Pour déterminer la borne supérieure du temps de réponse des tâches avec gigue sur activation, ils utilisent la formule établie dans [AUD 93] et l'adaptent en la formule de récurrence suivante :

$$R_i^0 = C_i$$
$$R_i^{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n + J_j}{T_j} \right\rceil C_j \tag{5.1}$$

La récurrence s'arrête au pas n, lorsque $R_i^{n+1} = R_i^n$.

Pour vérifier l'ordonnançabilité d'une tâche, il faut vérifier que le pire temps de réponse plus le délai de gigue soient inférieurs à l'échéance relative : $R_i^n + J_i \leq D_i$.

5.2.3. *Les travaux de* MING

MING *et al.*, [MIN 94] modifient la relation de récurrence de WELLINGS (formule 5.1) pour étudier le problème des tâches à suspension. Ainsi, par rapport à la formule 5.1, le paramètre de gigue disparaît et est remplacé par la durée de suspension. MING considère la durée de suspension comme partie intégrante de la durée d'exécution requise par la tâche. La formule de calcul de la borne supérieure du temps de réponse est la suivante :

$$R_i^0 = (C_i + X_i)$$
$$R_i^{n+1} = (C_i + X_i) + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n + X_j}{T_j} \right\rceil C_j \tag{5.2}$$

L'assertion de MING *et al.* qui considère le délai de suspension comme du temps processeur de la tâche, dénature les tâches à suspension puisque l'exécution des opérations externes se fait sur un processeur dédié. En conséquence, une telle approche augmente considérablement le pessimisme et ainsi directement le pire temps de réponse des tâches à suspension. KIM *et al.* (*cf.* [KIM 95]) définissent deux nouveaux tests d'ordonnançabilité pour calculer le pire temps de réponse des tâches à suspension sur la base de l'ensemble de ces précédents travaux.

Exemple 14 *Nous présentons maintenant un exemple d'utilisation de la méthode de* MING*, sur la configuration de tâches à suspension I. Les résultats de l'application de la formule 5.2 sont représentés par le tableau 5.1. Nous calculons récursivement chacune des valeurs jusqu'à ce qu'elles obtiennent des valeurs stationnaires.*

Ensuite pour chaque tâche, il suffit de vérifier que la borne supérieure du temps de réponse ainsi calculée soit inférieure à la période, ce qui est vrai pour toutes les tâches donc la configuration est ordonnançable par les algorithmes à priorité fixe d'après le test de MING.

Tâches	R_i^0	R_i^1	R_i^2	période	ordonnançable ?
τ_1	3	3		8	oui
τ_2	7	9	11	40	oui
τ_3	5	11	13	80	oui

Tableau 5.1. *Exemple d'application de la méthode de* MING

5.2.4. *La méthode A de* KIM

Cette méthode se base sur les travaux de WELLINGS (formule 5.1).

Pour leur modèle de tâches, ils considèrent tout d'abord que les échéances des tâches sont inférieures aux périodes : $D_i \leq T_i$ pour tout i ($1 \leq i \leq n$) et que les tâches sont préemptibles. Cette première méthode subdivise chaque tâche à suspension τ_i en deux tâches indépendantes et sans suspension après le début de leurs exécutions $\tau_{i,1}$ et $\tau_{i,2}$. Comme le montre la figure 5.4, les deux tâches indépendantes $\tau_{i,1}$ et $\tau_{i,2}$ héritent de la tâche τ_i, sa date d'activation, sa période et son échéance. De plus $\tau_{i,1}$ a pour temps processeur requis, le premier bloc d'exécution de τ_i à savoir $C_{i,1}$ et $\tau_{i,2}$, le second bloc $C_{i,2}$.

Pour s'assurer que la tâche $\tau_{i,1}$ s'exécute avant la tâche $\tau_{i,2}$, la priorité de la tâche $\tau_{i,1}$ doit être supérieure à celle de la tâche $\tau_{i,2}$. De plus, pour que la suspension soit respectée, une caractéristique supplémentaire est ajoutée aux tâches : la gigue sur activation (*cf.* paragraphe 4.3.1). Une valeur de gigue $J_{i,1} = 0$ pour la tâche $\tau_{i,1}$ et $J_{i,2} = X_i$ pour la tâche $\tau_{i,2}$.

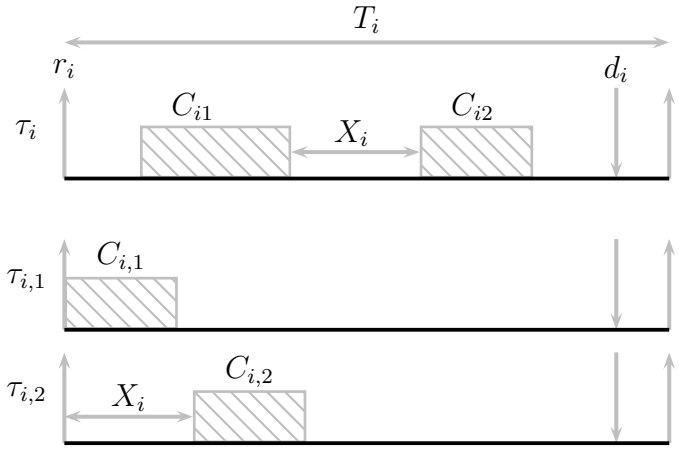

Figure 5.4. *Principe de décomposition de la méthode A de* KIM

La configuration de tâches nouvellement générées à partir de la configuration de tâches à suspension est une configuration de tâches indépendantes, sans suspension mais avec des gigues sur activation. Pour calculer la borne supérieure du temps de réponse des tâches, la formule 5.1 peut être utilisée. Ainsi, afin de déterminer l'ordonnançabilité d'une configuration, le déroulement du test est le suivant :

– chacune des tâches τ_i est subdivisée en deux tâches indépendantes $\tau_{i,1}$ et $\tau_{i,2}$;

– la borne supérieure du temps de réponse des tâches générées $\tau_{i,1}$ est calculée en utilisant la formule 5.1. Ce qui donne la formule suivante :

$$R_{i,1}^{n+1} = C_{i,1} + \sum_{j=1}^{i-1} \left\lceil \frac{R_{i,1}^{n}}{T_j} \right\rceil C_{j,1} + \sum_{j=1}^{i-1} \left\lceil \frac{R_{i,1}^{n} + X_j}{T_j} \right\rceil C_{j,2}$$

– La récurrence s'arrête au premier pas entier n satisfaisant $R_{i,1}^{n+1} = R_{i,1}^{n}$ et alors, le pire temps de réponse de $\tau_{i,1}$ noté $R_{i,1}^{*}$ et égal à $R_{i,1}^{n}$. Si $R_{i,1}^{*}$ est inférieur à D_i alors $\tau_{i,1}$ est ordonnançable par les algorithmes à priorité fixe. Sinon, aucune conclusion n'est possible quant à l'ordonnançabilité de $\tau_{i,1}$.

– De même la formule pour calculer la borne supérieure du temps de réponse des tâches $\tau_{i,2}$ est :

$$R_{i,2}^{n+1} = C_{i,2} + \sum_{j=1}^{i-1} \left\lceil \frac{R_{i,2}^{n}}{T_j} \right\rceil C_{j,1} + \sum_{j=1}^{i-1} \left\lceil \frac{R_{i,2}^{n} + X_j}{T_j} \right\rceil C_{j,2}$$

– La borne supérieure du pire temps de réponse $R_{i,2}^{*}$ de la tâche $\tau_{i,2}$ est calculée. Enfin, il reste à vérifier que l'échéance de τ_i est respectée : $(R_{i,1}^{*} + X_i + R_{i,2}^{*}) \leq D_i$, pour affirmer que la tâche à suspension τ_i est ordonnançable. Dans le cas contraire, il n'y a pas de conclusion possible.

Exemple 15 *Pour conclure la présentation de cette méthode, nous en montrons un exemple d'utilisation sur la configuration de tâches à suspension I. La méthode A de* KIM *transforme la configuration I en une configuration I' de tâches sans suspension. Les caractéristiques des tâches de la configuration I' sont présentées par les quatre premières colonnes du tableau 5.2. Ensuite, pour chaque tâche et en utilisant les formules établies, la borne supérieure*

Tâches	Gigue sur activation	Durée d'exécution	Période	Temps de réponse
$\tau_{1,1}$	0	1	8	1
$\tau_{1,2}$	1	1	8	1
$\tau_{2,1}$	0	3	40	5
$\tau_{2,2}$	3	1	40	3
$\tau_{3,1}$	0	1	80	7
$\tau_{3,2}$	2	2	80	10

Tableau 5.2. *Exemple d'application de la méthode A de* KIM

du temps de réponse est calculée. Nous détaillons dans le tableau 5.3, le calcul du temps de réponse de la tâche τ_2, en sachant que les temps de réponse des sous-tâches $\tau_{1,1}$ et $\tau_{1,2}$

sont égaux à 1 *et ce, dès le premier pas de récurrence (car la tâche* τ_1 *est la tâche la plus prioritaire de* I*).*

Tâche	R_2^0	R_2^1	R_2^2
$\tau_{2,1}$	$C_{2,1} = 3$	$C_{2,1} + \lceil \frac{3}{8} \rceil + \lceil \frac{4}{8} \rceil = 5$	5
$\tau_{2,2}$	$C_{2,2} = 1$	$C_{2,2} + \lceil \frac{1}{8} \rceil + \lceil \frac{2}{8} \rceil = 3$	3

Tableau 5.3. *Calcul du temps de réponse de la tâche* τ_2 *de* I *par la méthode A de* KIM

Finalement, en regroupant ces temps calculés ($\tau_{i,1}$ *et* $\tau_{i,2}$*), une borne supérieure du temps de réponse de chaque tâche* τ_i *de la configuration* I *est calculée* $R_i^* = R_{i,1}^* + X_i + R_{i,2}^*$ *:*

$$\tau_1 : R_1^* = 3$$

$$\tau_2 : R_2^* = 11$$

$$\tau_3 : R_3^* = 19$$

En comparant ces bornes supérieures sur le temps de réponse et les échéances relatives des tâches, nous constatons que toutes les tâches et donc la configuration sont ordonnançables par les algorithmes à priorité fixe.

5.2.5. *La méthode B de* KIM

Cette approche est une amélioration des travaux de MING (*cf.* formule 5.2). La méthode de MING considère les délais de suspension des tâches comme du temps processeur requis par la tâche. Cette méthode peut augmenter considérablement le temps de réponse des tâches ce qui rend son utilisation limitée. La seconde méthode de KIM réutilise ce principe mais diminue le temps dû à la suspension des tâches inclus dans le temps processeur. Le temps supprimé du délai de la suspension est celui correspondant à l'exécution des tâches plus prioritaires $\sum_{j=1}^{i-1} \left\lfloor \frac{X_i}{T_j} \right\rfloor C_j$. La formule de récurrence de la borne supérieure du calcul du pire temps de réponse est :

$$R_i^{n+1} = C_i + M_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_{j,1} + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n + X_j}{T_j} \right\rceil C_{j,2}$$

Où $M_i = X_i - \sum_{j=1}^{i-1} \left\lfloor \frac{X_i}{T_j} \right\rfloor C_j$

Dès que $R_i^{n+1} = R_i^n$ alors la récurrence est stoppée et de plus, si R_i^n est inférieur à D_i alors la tâche à suspension τ_i est ordonnançable. Dans le cas contraire, il n'y a pas de conclusion possible sur l'ordonnançabilité de la tâche.

Remarque 9 *Comme M_i est inférieur à X_i, si la tâche à suspension τ_i est ordonnançable avec la méthode de* MING *(cf. Formule 5.2), alors elle le sera aussi avec le test d'ordonnançabilité de la méthode B de* KIM.

Exemple 16 *Exemple d'application de la méthode B de* KIM *sur la configuration de tâches à suspension I. La borne supérieure du temps de réponse de chaque tâche est calculée en utilisant la formule décrite ci-dessus. Afin de s'assurer l'ordonnançabilité de chaque tâche, il faut vérifier que sa borne supérieure du temps de réponse soit inférieure à sa période. Le calcul de la borne supérieure du temps de réponse de la tâche τ_2 est présenté par le tableau 5.4.*

Tâche	M_2	R_2^0	R_2^1	R_2^2	R_2^3
τ_2	3	$C_2 + M_2 = 7$	$7 + \lceil \frac{7}{8} \rceil + \lceil \frac{8}{8} \rceil = 9$	$7 + \lceil \frac{9}{8} \rceil + \lceil \frac{10}{8} \rceil = 11$	11

Tableau 5.4. *Calcul du temps de réponse de la tâche τ_2 de I par la méthode B de* KIM

L'ordonnançabilité de la configuration I est représentée dans le tableau 5.5.

Tâches	Temps de réponse	Période	Ordonnançabilité ?
τ_1	3	8	oui
τ_2	11	40	oui
τ_3	13	80	oui

Tableau 5.5. *Exemple d'application de la méthode B de* KIM

Les trois tâches de la configuration sont ordonnançables avec les algorithmes à priorité fixe donc la configuration de tâches à suspension I également.

5.2.6. *La méthode de* LIU *[LIU 00]*

Pour prendre en compte les suspensions des tâches, LIU (*cf.* [LIU 00]) introduit dans son test d'ordonnançabilité, un facteur de blocage. Pour prendre en compte le délai supplémentaire que la tâche τ_i subit à cause de sa propre suspension et des suspensions des tâches plus prioritaires, LIU [LIU 00] représente ce délai par un facteur de blocage qu'il faut ajouter au calcul du pire temps de réponse. Ce facteur est dénoté $b_i(ss)$ (*ss* pour ***Self-Suspension*** dans [LIU 00]).

Le temps de blocage d'une tâche à suspension τ_i dû à sa propre suspension est, par construction, inférieur à X_i, la durée maximum de suspension. Pour déterminer le temps de blocage subi par une tâche τ_i et dû aux tâches plus prioritaires, deux points sont à considérer :

– Une tâche τ_k plus prioritaire que τ_i, ne peut pas retarder la tâche τ_i pendant plus de C_k unités de temps puisque la tâche τ_k peut être ordonnancée (au moins partiellement) pendant la suspension de la tâche τ_i parce que le processeur est libre.

– Par ailleurs, si le délai de suspension X_k est inférieur à C_k alors le temps de blocage subi par la tâche τ_i à cause de la tâche τ_k ne peut pas dépasser les X_k unités de temps.

En conclusion, le facteur de blocage subi par une tâche et dû à une tâche plus prioritaire τ_k, n'est jamais supérieur au délai de suspension de τ_k et ne dépasse jamais C_k unités de temps. Ainsi il est égal à :

$$\min(C_k, X_k)$$

Le temps de blocage d'une tâche τ_i à cause de sa propre suspension et des délais de suspension des tâches plus prioritaires est égal à :

$$b_i(ss) = X_i + \sum_{k=1}^{i-1} \min(C_k, X_k)$$

Finalement, en utilisant la formule itérative [AUD 93], pour calculer la borne supérieure du temps de réponse, il faut calculer successivement sur n, la valeur de la formule suivante :

$$R_i^{n+1} = C_i + b_i(ss) + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j$$

L'itération est stoppée au rang n dès que $R_i^n = R_i^{n+1}$, et la borne supérieure est égale à $R_i^* = R_i^n$. Il ne reste plus qu'à vérifier que $R_i^* \leq D_i$ pour vérifier l'ordonnançabilité de la tâche τ_i. Dans le cas contraire, aucune conclusion ne peut être tirée.

Exemple 17 *Exemple d'application de la méthode de* LIU *sur la configuration de tâches à suspension I. Il faut, pour se faire dans un premier temps évaluer le temps de blocage dû à la suspension de la tâche et à celle des tâches plus prioritaires. Ces temps de blocage sont donnés dans la seconde colonne. Ensuite, la borne supérieure du temps de réponse de chaque tâche de la configuration (troisième colonne) est calculée. Le tableau 5.6 représente le calcul de la borne supérieure du temps de réponse de la tâche τ_2 de la configuration I.*

Puis finalement, nous pouvons vérifier que ce temps calculé est inférieur à la période (quatrième colonne) et ainsi déterminer l'ordonnançabilité de chaque tâche de la configuration (dernière colonne).

Les trois tâches de I sont ordonnançables avec les algorithmes à priorité fixe donc la configuration de tâches à suspension I également.

Tâche	$b_2(ss)$	R_2^0	R_2^1	R_2^2
τ_2	4	$C_2 + b_2(ss) = 8$	$8 + \lceil \frac{8}{8} \rceil 2 = 10$	$8 + \lceil \frac{9}{8} \rceil 2 = 12$

Tableau 5.6. *Calcul du temps de réponse de la tâche τ_2 de I par la méthode de* LIU

Tâches	Temps de blocage	Temps de réponse	Période	Ordonnançabilité ?
τ_1	1	3	8	oui
τ_2	4	12	40	oui
τ_3	6	21	80	oui

Tableau 5.7. *Exemple d'application de la méthode de* LIU

5.3. Conclusion

Nous avons présenté plusieurs tests d'ordonnançabilité concernant les configurations de tâches à suspension. Il existe d'autres tests d'ordonnançabilité. Notamment, celui de PALENCIA et al. [PAL 98] basé sur le calcul de dates d'activations dynamiques (*dynamic offsets*) pour les algorithmes à priorité fixe ; ce test est décrit en annexe (*cf.* annexe A). PALENCIA et al. ont aussi établi un autre test d'ordonnançabilité pour EDF, [PAL 03], rapporté en annexe B. Ces tests ne seront pas considérés dans la suite car le modèle de tâches est différent (transaction de tâches) d'une part, et celles-ci sont activées sur des évènements d'autre part (ce qui a des conséquences sur le pire scénario d'activation). Finalement, un dernier test d'ordonnançabilité, établi pour l'algorithme EDF est consigné dans l'annexe C, algorithme de DEVI, [DEV 03].

5.4. Bibliographie

[AUD 93] AUDSLEY N., BURNS A., RICHARDSON M., TINDELL K., WELLINGS A., « Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling », *Software Engineering Journal*, vol. 8, n°5, p. 284-292, 1993.

[BRU 01] BRUCKER P., *Scheduling Algorithms,* (Third edition) Springer Verlag, 2001.

[BUR 95] BURNS A., « Preemptive Priority-Based Scheduling : An Appropriate Engineering Approach », *in Advances in Real-Time Systems, S.H. Son, Ed., Prentice Hall, New Jersey*, p. 225-248, 1995.

[DEV 03] DEVI U., « An Improved Schedulability Test for Uniprocessor Periodic Task Systems », *IEEE Euromicro Conf. on Real-Time Systems (ECRTS'03)*, p. 23-32, 2003.

[KIM 95] KIM I., CHOI K., PARK S., KIM D., HONG M., « Real-Time Scheduling of tasks that contain the external blocking intervals », *proc. Conference on Real-Time Computing Systems and Applications*, p. 54-59, 1995.

[LIU 00] LIU J., *Real-Time Systems,* Prentice hall, 2000.

[MIN 94] MING L., « Scheduling of the Inter-Dependent Messages in Real-Time Communication », *Proc. of the First International Workshop on REal-Time Computing Systems and Applications*, décembre 1994.

[PAL 98] PALENCIA J., GONZALES-HARBOUR M., « Schedulability Analysis for Tasks with Static and Dynamic Offsets », *Proceedings of the* 19^th *IEEE Real-Time Systems Symposium*, 1998.

[PAL 03] PALENCIA J., GONZALES-HARBOUR M., « Offset-Based Response Time Analysis of Distributed Systems Scheduled under EDF », *proc IEEE Euromicro Conf. on Real-Time Systems (ECRTS'03)*, page , 2003.

Chapitre 6

Difficultés de l'ordonnancement de tâches à suspension

6.1. Introduction

Les tâches à suspension sont des tâches qui au cours de leur exécution doivent se suspendre pendant un certain temps, le temps d'effectuer une opération externe sur un processeur annexe. Le but de ce chapitre est d'étudier les difficultés de l'ordonnancement de configurations de tâches à suspension.

La difficulté d'un problème d'ordonnancement se détermine suivant trois principaux axes :

– la complexité : la première caractéristique importante à déterminer est la complexité du problème d'ordonnancement. En effet, avant de concevoir un algorithme pour un problème d'ordonnancement, il est fondamental de connaître à quelle classe de complexité il appartient.

– la présence d'anomalies (paragraphe 4.3.4) : la possible présence d'anomalies pour un problème d'ordonnancement rend plus complexe la caractérisation du pire comportement du système pouvant conduire à ne pas respecter des échéances de tâches.

– l'optimalité : elle permet de savoir pour le problème d'ordonnancement étudié s'il peut exister un algorithme en-ligne qui soit optimal pour le critère étudié.

Ce chapitre se décompose en trois paragraphes principaux. Dans le paragraphe 6.2, nous démontrons que la complexité du problème d'ordonnancement de tâches à suspension est $\mathcal{NP}$-Difficile au sens fort. Le paragraphe 6.3 établit la présence d'anomalies pour l'ordonnancement de tâches à suspension. Finalement, le paragraphe 6.4 prouve la non optimalité de n'importe quel algorithme en-ligne déterministe.

6.2. Complexité

Dans ce paragraphe, nous faisons apparaître que le problème d'ordonnancement de tâches périodiques qui peuvent se suspendre au plus une fois au cours de leur exécution, est un problème $\mathcal{NP}$-Difficile au sens fort. Il a déjà été démontré dans [RIC 03] que ce problème est $\mathcal{NP}$-Difficile au sens fort pour l'ordonnancement de systèmes de tâches périodiques, à

départ simultané et échéances contraintes ($D_i \leq T_i$). Dans ce paragraphe, nous étudions le problème ouvert de complexité où les tâches sont à échéance sur requête et quand elles ne peuvent se suspendre qu'au plus une fois. De plus, nous démontrons qu'il n'existe pas d'algorithme *universel* pour ordonnancer des tâches avec suspension, sauf si $\mathcal{P} = \mathcal{NP}$.

Theorème 5 *L'ordonnancement de tâches périodiques, à échéance sur requête, à départ simultané et se suspendant au cours de leur exécution au plus une fois est un problème $\mathcal{NP}$-Difficile au sens fort.*

Démonstration :

Pour démontrer ce théorème, nous allons transformer une instance du problème de 3-partition connu pour être $\mathcal{NP}$-Complet au sens fort en une instance de notre problème [GAR 79]. Enfin, nous essayerons d'ordonnancer cette configuration générée.

Instance du problème de 3-partition : Soit A un ensemble de $3m$ éléments de $\mathbb{N}$, B une limite appartenant à $\mathbb{N}$ et pour tout $j \in \{1 \dots 3m\}$, $s_j \in \mathbb{N}$ tels que $B/4 < s_j < B/2$ et $\sum_{i=1}^{3m} s_j = mB$.

Problème : Peut-on partitionner A en m ensembles disjoints (A_1, A_2, ..., A_m) tels que, pour $1 \leq i \leq m$, $\sum_{j \in A_i} s_j = B$? Ce qui implique que chaque ensemble A_i, $i \in \{1, \dots, n\}$ contient exactement trois éléments. En effet, il ne peut ni en contenir moins car $s_j < B/2$ et ni en contenir plus car $B/4 < s_j$.

Nous générons à partir de l'instance du problème 3-partition, une instance d'ordonnancement à $3m + 1$ tâches :

– Les tâches $\tau_1, \dots, \tau_{3m}$ sont générées avec le même profil :

$$i \leq 3m \quad \begin{cases} C_{i,1} = C_{i,2} &= s_i \\ X_i &= (2m-1)B \\ D_i = T_i &= 4mB \end{cases}$$

– La tâche τ_{3m+1} :

$$C_{3m+1,1} = \left\lceil \frac{B}{2} \right\rceil \qquad C_{3m+1,2} = \left\lfloor \frac{B}{2} \right\rfloor$$

$$X_{3m+1} = B$$

$$D_{3m+1} = T_{3m+1} = 2B$$

Nous allons maintenant prouver qu'une solution pour le problème de 3-partition existe si et seulement si la configuration de tâches générée à partir de ce problème est ordonnançable.

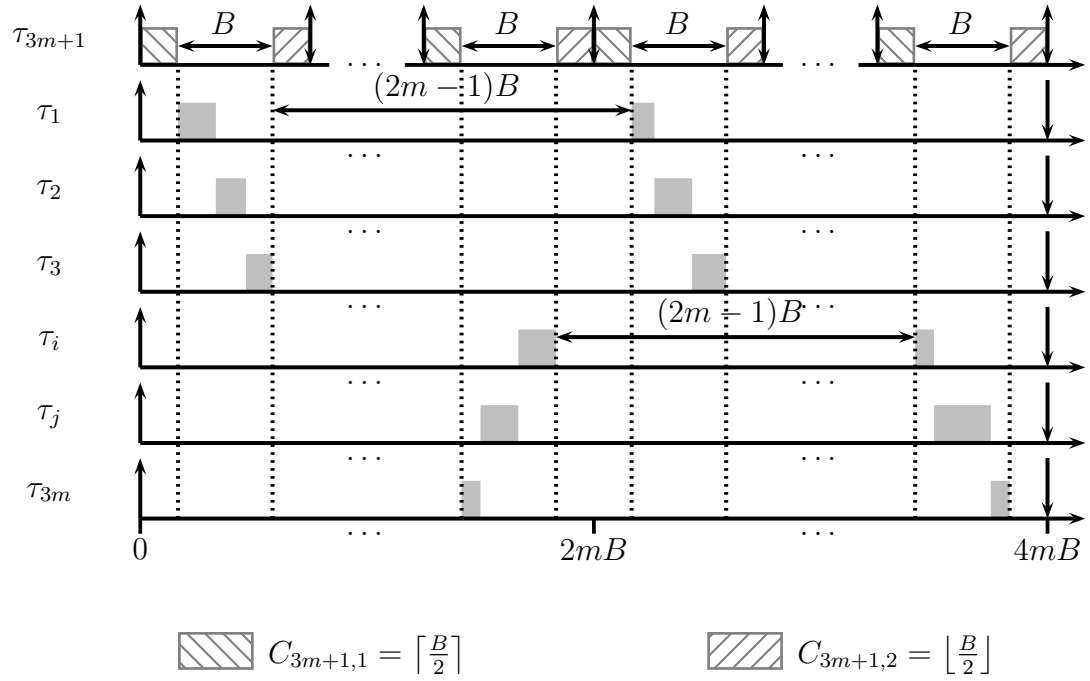

Figure 6.1. *Ordonnancement faisable de la configuration générée*

Les tâches générées sont toutes à départ simultané. Par conséquent, nous pouvons limiter notre étude d'ordonnançabilité à l'hyperpériode dont la longueur est égale à :
$H = ppcm(T_1, \ldots, T_{3m+1}) = 4mB$. Nous limiterons notre travail à l'intervalle $[0, 4mB]$.

Nous pouvons également montrer que le facteur d'utilisation de cette configuration est égal à 1 :

– Le facteur d'utilisation des tâches $\tau_1, \ldots, \tau_{3m}$ est égal à :

$$\begin{aligned} \sum_{i=1}^{3m} \frac{C_{i,1}+C_{i,2}}{T_i} &= \sum_{i=1}^{3m} 2\frac{s_i}{4mB} \\ &= \frac{2mB}{4mB} \\ &= \frac{1}{2} \end{aligned}$$

– Le facteur d'utilisation de τ_{3m+1} vaut :

$$\begin{aligned} \frac{C_{3m+1,1}+C_{3m+1,2}}{T_{3m+1}} &= \frac{\lceil \frac{B}{2} \rceil + \lfloor \frac{B}{2} \rfloor}{2B} \\ &= \frac{B}{2B} \\ &= \frac{1}{2} \end{aligned}$$

Le facteur d'utilisation de cette configuration est par conséquent bien égal à $\frac{1}{2} + \frac{1}{2} = 1$. Ce résultat implique que le processeur est constamment occupé et que par conséquent, il n'y a aucun temps libre dans l'ordonnancement. La tâche τ_{3m+1} possède une laxité nulle, par

conséquent, chacune de ses instances doit être exécutée dès son arrivée dans le système. La tâche τ_{3m+1} laisse libre le processeur pendant seulement, pour chacune de ses instances, un unique bloc de durée B (la longueur de la suspension) pour l'exécution des autres tâches. La figure 6.1 représente un ordonnancement réussi de la configuration générée par le problème de 3-partition. Les bornes du $k^{\text{ième}}$ bloc de temps creux sont calculables et sont les suivantes :

$$\left[2(k-1)B + \left\lceil \frac{B}{2} \right\rceil ; 2kB - \left\lfloor \frac{B}{2} \right\rfloor \right) \qquad \forall k \geq 1 \tag{6.1}$$

Nous allons aborder notre démonstration en deux parties.

1) **Si une solution existe pour le problème de 3-partition alors la configuration de tâches générée admet un ordonnancement faisable :**
D'après l'énoncé du problème et si nous nous intéressons à A_1, il possède exactement trois éléments (notés s_j, s_k et s_l) dont la somme ($s_j + s_k + s_l$) donne B. Pendant le temps creux correspondant à la suspension de la première instance de τ_{3m+1}, nous ordonnançons les premières sous-tâches ($C_{i,1}$) des tâches (τ_j, τ_k et τ_l) correspondant aux éléments de A_1. Les secondes sous-tâches sont ordonnancées pendant le temps creux créé par la $(m+1)^{\text{ième}}$ instance de τ_{3m+1}. Ainsi le temps écoulé entre les deux sous-tâches de chaque tâche générée à partir des éléments de A_1 est égal à $(2m-1)B$, soit la durée de suspension requise d'après la formule 6.1. Nous recommençons le même processus en prenant les éléments de A_2 et en réitérant l'opération en ordonnançant les premières sous-tâches des tâches générées par les éléments de A_2 pendant le temps creux généré par la seconde instance de τ_{3m+1} et les secondes sous-tâches pendant le temps creux obtenu par l'exécution de la $(m+2)^{\text{ième}}$ instance de τ_{3m+1}. Puis, il suffit de continuer les mêmes opérations en ordonnançant les tâches correspondant aux éléments A_i pendant les $i^{\text{ième}}$ et $(m+i)^{\text{ième}}$ temps creux générés par les instances de τ_{3m+1}. Ainsi, nous obtenons un ordonnancement faisable pour notre configuration de tâches.

2) **Si la configuration de tâches générées admet un ordonnancement faisable alors il existe une solution pour le problème de 3-partition :** Nous allons d'abord supposer qu'il existe un ordonnancement non préemptif pour les tâches générées puis nous traiterons le cas général.

- Il existe un ordonnancement non préemptif pour les tâches générées à partir du problème de 3-partition :
Comme le facteur d'utilisation est de 1 et que l'ordonnancement est faisable, dans chaque temps creux dû à l'exécution d'une requête de τ_{3m+1} trois tâches exécutent soit leur première sous-tâche, soit leur seconde (elles sont trois car $B/4 < C_{i,j} < B/2, i = 1..3m, j = 1, 2$). Les trois tâches, dont la première sous-tâche s'est exécutée pendant le temps creux dû à l'exécution de la première requête de la tâche τ_{3m+1}, ne peuvent pas exécuter leur seconde sous-tâche pendant les $m-1$ blocs de temps creux suivants générés par τ_{3m+1} sinon la durée de suspension n'est pas respectée. Par conséquent, la seconde sous-tâche ne peut être ordonnancée qu'à partir du $(m+1)^{\text{ième}}$ bloc de temps creux. Pendant ce temps creux, toutes les premières sous-tâches de toutes les tâches τ_i (pour $i \in \{1 \dots 3m\}$) ont été exécutées et toutes ces tâches sont en période de suspension sauf celles qui ont exécuté leur première sous-tâche durant le premier bloc de temps creux. Par conséquent, comme le facteur d'utilisation

est de 1, pendant le $(m+1)^{\text{ième}}$ bloc de temps creux, les seules tâches ordonnancées sont celles dont la première sous-tâche a été exécutée durant le premier bloc de temps creux. En répétant cette logique pour les autres tâches, les tâches qui ont exécuté leur première sous-tâche pendant le $i^{\text{ième}}$ ($i \leq m$) bloc de temps creux exécutent leur seconde sous-tâche pendant le $(m+i)^{\text{ième}}$ bloc de temps creux. Ainsi, si nous nommons A_i, l'ensemble des s_k ayant servi à la conception des tâches dont la première sous-tâche a été exécutée pendant le $i^{\text{ième}}$ temps creux généré par τ_{3m+1}, alors nous obtenons une partition de A en m sous-ensembles respectant les contraintes données par le problème de 3-partition.

- Cas général :

Nous allons en fait démontrer qu'une sous-tâche ne peut s'exécuter que dans un unique bloc de temps creux généré par τ_{3m+1}. En effet, utilisons un raisonnement par l'absurde et supposons qu'une sous-tâche ait commencé son exécution dans un bloc k ($k < m$) et qu'elle finisse son exécution dans le bloc $k+1$. Les autres sous-tâches sont supposées s'être exécutées dans un unique bloc. Comme il ne peut y avoir plus de trois sous-tâches par bloc, il n'y a que deux sous-tâches dont l'exécution s'est terminée dans le bloc k. Par respect des suspensions des tâches, il n'y a seulement que deux sous-tâches complétées dans le bloc $m+k$. En conséquence, il y a un temps creux dans le bloc $k+m$ ce qui contredit le fait que le facteur d'utilisation est égal à 1. Ce qui met un point final à cette démonstration.

□

Définition 19 *Un algorithme d'ordonnancement est dit* universel *si cet algorithme effectue le choix de la prochaine tâche à ordonnancer en temps polynomial [JEF 91].*

Nous démontrons maintenant qu'il n'existe pas d'algorithme *universel* pour ordonnancer des tâches avec suspension, à moins que $\mathcal{P} = \mathcal{NP}$.

Theorème 6 *S'il existe un algorithme d'ordonnancement universel pour les configurations de tâches où chaque tâche se suspend au plus une fois alors $\mathcal{P} = \mathcal{NP}$.*

Démonstration :

Nous allons utiliser une technique de preuve classique, telle que celle présentée dans [JEF 91]. Plus précisément, nous allons supposer l'existence d'un tel algorithme. Et nous allons démontrer que si cet algorithme choisit en un temps polynomial (polynomial en la longueur de la configuration) la prochaine tâche exécutée, alors $\mathcal{P} = \mathcal{NP}$. Parce que dans ce cas-là, nous aurions trouvé un algorithme pseudo-polynomial capable de résoudre le problème de 3-partition.

Nous supposons qu'il existe un algorithme d'ordonnancement concernant les configurations de tâches périodiques, où chaque tâche peut se suspendre au plus une fois, et sur

des systèmes monoprocesseurs. Cet algorithme est noté A. En utilisant la même technique que celle détaillée dans la preuve du théorème 5, nous définissons à partir d'une instance du problème de 3-partition un ensemble de $3m + 1$ tâches, noté I. L'ensemble des tâches composant la configuration nouvellement générée I est à départ simultané. En utilisant le même raisonnement que dans la démonstration du théorème 5, pour vérifier que toutes les instances respectent leurs échéances, nous pouvons limiter notre période d'étude à l'intervalle $[0, 4Bm]$. Par conséquent, nous ordonnançons la configuration I avec l'algorithme A, puis nous vérifions que chaque tâche respecte son échéance. L'hyperpériode de l'ordonnancement, comme nous l'avons calculée précédemment, est égale à $4Bm$. De plus, d'après nos hypothèses, A est un algorithme polynomial. Par conséquent, vérifier que toutes les échéances sont respectées se fait en un temps au plus pseudo-polynomial (c.-à-d. s'est clairement réalisé en temps proportionnel à Bm). En utilisant le même raisonnement que lors de la démonstration du théorème 5, la configuration I n'est ordonnançable par l'algorithme A que si et seulement si, il existe une partition des tâches $\tau_1, \ldots, \tau_{3m}$ en m ensembles disjoints $A_1, A_2, \ldots, A_m$ tels que pour chaque ensemble A_i ($i \in \{1, \ldots, m\}$), nous avons $\sum_{\tau_j \in A_i} C_{j,1} = B$. Ainsi, la solution obtenue par l'algorithme A donne une solution pour résoudre le problème de 3-partition. Afin de déterminer cette solution, nous transformons l'instance du problème de 3-partition en construisant simplement l'ensemble de tâches avec au plus une suspension par tâche comme dans la preuve du théorème 5 et alors en présentant cette configuration de tâches à la procédure de décision basée sur l'algorithme A.

Par conséquent, nous avons trouvé un algorithme pseudo-polynomial pour résoudre le problème de 3-partition. Mais le problème de 3-partition est $\mathcal{NP}$-complet au sens fort et ne peut pas en conséquence être résolu par un algorithme pseudo-polynomial sauf si $\mathcal{P} = \mathcal{NP}$. Donc, et pour conclure, si l'algorithme A existe alors $\mathcal{P} = \mathcal{NP}$. Un tel algorithme ne peut donc pas par conséquent exister.

□

6.3. Anomalies d'ordonnancement

Dans cette sous-partie, nous allons étudier les possibilités d'apparition d'anomalies sous la politique d'ordonnancement *E*arliest-*D*eadline-*F*irst (EDF). Nous explicitons tout d'abord la notion d'*anomalie d'ordonnancement*. Soit A un algorithme, nous savons que le temps processeur requis pour A peut varier d'une exécution à une autre. A possède des anomalies d'ordonnancement, s'il existe une configuration I de tâches telle que diminuer la durée d'exécution ou de suspension d'une des tâches rend la configuration I non ordonnançable par A, alors qu'elle l'était en considérant les pires durées d'exécution et de suspension de l'ensemble des tâches de la configuration. Un algorithme supportant les anomalies d'ordonnancement est dit *robuste*. En pratique, la robustesse simplifie les tests d'ordonnnaçabilité puisqu'il est suffisant de ne considérer que les pires durées d'exécution des tâches.

Il a été prouvé dans [LIU 00] qu'*EDF* est robuste pour l'ordonnancement de tâches indépendantes sans suspension. Ainsi, dans un ordonnancement, toutes les dates limites de fin

d'exécution sont respectées avec les pires durées d'exécution ; diminuer la durée d'exécution d'une des tâches ne peut pas amener *EDF* à ne plus respecter l'échéance d'une des tâches du système. Nous allons démontrer que cette assertion est fausse dès lors que des tâches sont autorisées à se suspendre.

Theorème 7 *Des anomalies d'ordonnancement peuvent apparaître en exécutant des tâches à suspension par l'algorithme d'ordonnancement EDF.*

Démonstration :

Afin de démontrer ce théorème, une configuration de tâches I va être définie. Puis sur cette configuration, il va être prouvé que si la durée d'exécution ou la durée de suspension d'une des tâches est réduite d'une unité de temps, une échéance ne sera plus respectée. Soit I la configuration suivante à trois tâches :

$$\tau_1 : r_1 = 0, C_{1,1} = 2, X_1 = 2, C_{1,2} = 2, D_1 = 6, T_1 = 10$$
$$\tau_2 : r_2 = 5, C_{2,1} = 1, X_2 = 1, C_{2,2} = 1, D_2 = 4, T_2 = 10$$
$$\tau_3 : r_3 = 7, C_{3,1} = 1, X_3 = 1, C_{3,2} = 1, D_3 = 3, T_3 = 10$$

Lorsque toutes les tâches sont exécutées avec leurs pires durées d'exécution et de suspension, *EDF* définit l'ordonnancement suivant : à l'instant 0, il ordonnance la tâche τ_1 jusqu'à l'instant 2. La tâche τ_1 se suspend de l'instant 2 à 4. À l'instant 4, elle redémarre son exécution et la termine à l'instant 6. À ce moment-là, la tâche τ_2, qui s'est réveillé à l'instant 5, s'exécute jusqu'à l'instant 7. Moment auquel elle se suspend et auquel la tâche τ_3 se réveille et alors commence son exécution. À l'instant 8, la tâche τ_3 se suspend et alors la tâche τ_2 qui ayant fini sa suspension peut achever son exécution à l'instant 9. La tâche τ_3, se réveillant à ce même instant, peut terminer elle-même son exécution à l'instant 10. Cet ordonnancement est représenté sur la figure 6.2. L'ordonnancement se déroule correctement et toutes les échéances sont respectées. Par conséquent, en utilisant les pires durées de suspension et d'exécution, l'ordonnancement de I par l'algorithme d'ordonnancement en-ligne *EDF* est faisable.

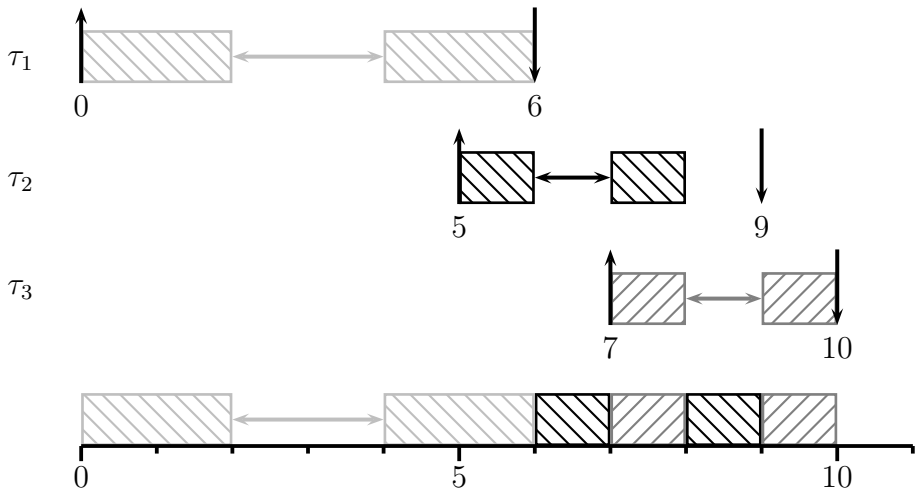

Figure 6.2. *Ordonnancement de la configuration I avec ses pires durées d'exécution des tâches*

Maintenant, en diminuant la durée d'exécution de la tâche τ_1 ou sa durée de suspension d'une unité de temps, il va être démontré que la tâche τ_3 ne respecte plus son échéance. Par exemple, diminuons la durée d'exécution $C_{1,1}$ d'une unité de temps : $C_{1,1} = 1$ et laissons toutes les autres durées d'exécution et de suspension inchangées. Comme τ_1 a une unité de moins d'exécution, elle termine par conséquent son exécution une unité plus tôt donc à l'instant 5. Mais à cet instant, τ_2 se réveille et peut par conséquent commencer son exécution une unité plus tôt. Donc à l'instant 7 quand elle se réveille après sa suspension, elle entre en concurrence avec la tâche τ_3 qui s'active. Cette concurrence n'existait pas au préalable puisque lorsque toutes les tâches sont exécutées avec leurs pires durées d'exécution et de suspension, la tâche τ_2 se suspend à la date 7 et ne demande donc pas à s'exécuter. La tâche τ_2, disposant d'une échéance plus petite que celle de la tâche τ_3, peut s'exécuter et finir son exécution à l'instant 8. Mais de ce fait, la tâche τ_3 se retrouve retardée d'une unité de temps et en conséquence elle ne respecte plus son échéance (cf. Figure 6.3.a), ce qui provoque une anomalie d'ordonnancement. Ainsi, en diminuant la durée d'exécution de la première sous-tâche de τ_1, l'ordonnancement est devenu irréalisable. Maintenant, réduisons à son tour la durée de suspension de la tâche τ_1 d'une unité de temps et laissons les autres paramètres inchangés. L'ordonnancement obtenu est représenté par la figure 6.3.a. La tâche τ_1 finit son ordonnancement à l'instant 5, à cet instant la tâche τ_2 est exécutée. τ_2 finit son ordonnancement à l'instant 8, ce qui décale l'exécution de τ_3 et lui fait manquer son échéance à l'instant 10. Si maintenant, la durée d'exécution $C_{1,2}$ est réduite d'une unité de temps tandis que les autres paramètres restent inchangés, la tâche τ_3 manque encore son échéance pour les mêmes raisons que celles précédemment citées.

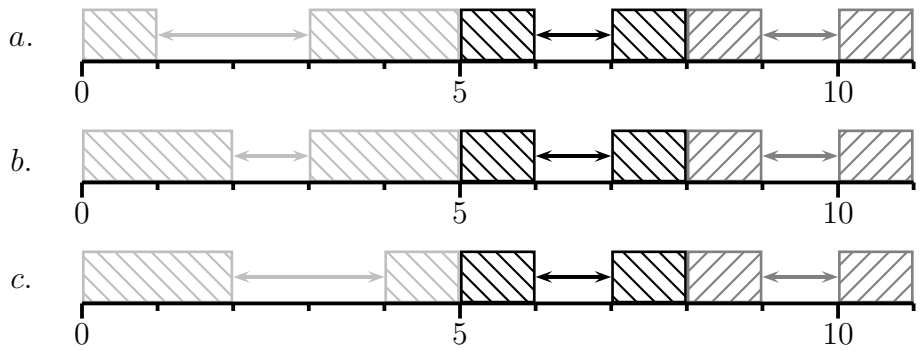

Figure 6.3. *Apparition d'anomalies d'ordonnancement en modifiant les caractéristiques de la tâche τ_1 de la configuration I*

Enfin, pour ces configurations où l'ordonnancement selon EDF n'est pas faisable, il existe un ordonnancement (hors-ligne) faisable. Par exemple, si nous reprenons la configuration de tâches I et que nous diminuons $C_{i,1}$ d'une unité de temps, nous avons démontré que l'ordonnancement de I par EDF est impossible. Mais si un algorithme décide, à l'instant 7 de retarder l'ordonnancement de la tâche τ_2 d'une unité de temps pour ordonnancer τ_3, celle-ci respecte alors son échéance tout comme les autres tâches ainsi que le montre la figure 6.4.

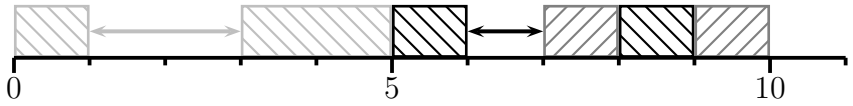

Figure 6.4. *Ordonnancement faisable de la configuration I, non ordonnançable par EDF*

□

Corollaire 1 *Pour les algorithmes à priorité fixe (cf.* RM *et* DM*), des anomalies d'ordonnancement peuvent apparaître lors de l'exécution de tâches à suspension.*

Démonstration :

Pour cette démonstration, nous reconsidérons la configuration de tâches à suspension définie dans la preuve du théorème 7. Sur cette configuration, les priorités affectées aux tâches par RM ou DM sont les mêmes que celles qu'affecte EDF. Par conséquent, les mêmes séquences d'exécution sont obtenues et donc les conclusions seront les mêmes que celles obtenues pour EDF.

□

6.4. Optimalité des algorithmes en-ligne

Dans ce paragraphe, nous démontrons qu'il n'existe pas d'algorithme en-ligne optimal pour l'ordonnancement de configurations de tâches sporadiques et à suspension sur des systèmes monoprocesseurs.

Theorème 8 *Aucun algorithme en-ligne déterministe n'est optimal pour l'ordonnancement de configurations de tâches sporadiques et à suspension sur des systèmes monoprocesseurs.*

Démonstration :

Pour démontrer ce théorème, nous utilisons l'analyse de compétitivité avec un adversaire évolutif (*cf.* paragraphe 2.5.3). Par conséquent, la démarche est de définir une configuration de tâches. Puis, selon les réactions de l'algorithme en-ligne en ordonnançant cette configuration de tâches à suspension, l'adversaire évolutif ajoute (ou pas) de nouvelles tâches à la configuration pour que la configuration finale soit non ordonnançable par l'algorithme en-ligne, alors qu'elle le sera de façon optimale par l'adversaire.

Ainsi, définissons la configuration de tâches à suspension I et montrons qu'aucun algorithme en-ligne ne peut ordonnancer optimalement I. La configuration que nous construisons comporte dans un premier temps deux premières tâches qui arrivent dans le système à l'instant 0 :

$$\tau_1 : C_{1,1} = 1, X_1 = 7, C_{1,2} = 1, D_1 = 10$$
$$\tau_2 : C_{2,1} = 1, X_2 = 4, C_{2,2} = 1, D_2 = 9$$

Soit A un algorithme en-ligne. À l'instant 0, l'algorithme A dispose de deux possibilités pour ordonnancer I : soit il décide d'ordonnancer τ_2, soit il décide de retarder son exécution.

1) **L'algorithme en-ligne A décide de ne pas ordonnancer τ_2 à l'instant 0 :**

- *l'algorithme en-ligne* A n'ordonnance pas τ_2 à l'instant 0 (cet ordonnancement est représenté par la figure 6.5.a). Il y a alors deux possibilités : ou l'algorithme A ordonnance la tâche τ_1 à l'instant 0 ou il laisse le processeur inoccupé et attend avant de commencer l'exécution d'une tâche. Mais dans les deux cas, il doit ordonnancer la tâche τ_2 à un instant t tel que $0 < t \leq 3$ pour respecter l'échéance de τ_2 (Pour la figure 6.5.a, nous avons pris t égal à 2). Dans ce cas, à l'instant 3 une nouvelle tâche (la tâche τ_3) s'active :

$$\tau_3 : C_{3,1} = 1, X_3 = 2, C_{3,2} = 3, D_3 = 9$$

À l'instant 3, l'algorithme en-ligne A ordonnance la tâche τ_3 puisqu'elle ne dispose pas de laxité, son ordonnancement doit être immédiat sous peine de manquer son échéance. Or en prenant cette décision l'algorithme en-ligne A ne dispose plus d'assez de temps pour compléter l'exécution des tâches τ_2 et τ_3 avant leur échéance commune à l'instant 9. Par conséquent, le choix de l'algorithme A est un mauvais choix et donc, l'ordonnancement de la configuration de tâches à suspension I par l'algorithme en-ligne A n'est pas faisable.

- *l'adversaire optimal hors-ligne* ordonnance à l'instant 0, la tâche τ_2 et à l'instant 1, la tâche τ_1. À l'instant 3, la tâche τ_3 arrive dans le système et est immédiatement ordonnancé par l'adversaire. À l'instant 5, la tâche τ_2 qui finit sa suspension est alors ordonnancée et termine son exécution à l'instant 6. Finalement, la tâche τ_3 achève son exécution à l'instant 9 et τ_1 à l'instant 10. Cet ordonnancement est représenté par la figure 6.5.b. Ainsi, même si l'ordonnancement de I par l'algorithme en-ligne A n'est pas faisable, cet ordonnancement par l'adversaire est quant à lui réalisable.

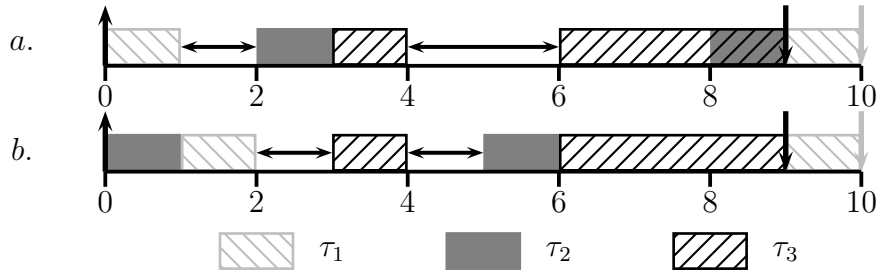

Figure 6.5. *Ordonnancement de I par un algorithme en-ligne A*
L'algorithme en-ligne n'ordonnance pas la tâche à suspension τ_2 à l'instant 0
mais à l'instant $t = 2$

Pour conclure ce premier cas, si un algorithme en-ligne choisit de ne pas ordonnancer la tâche τ_2 dès l'instant 0, l'ordonnancement de la configuration I n'est pas faisable alors qu'il existe un ordonnancement réalisable de I (l'adversaire). Ainsi l'algorithme en-ligne A n'est alors pas optimal.

2) **L'algorithme en-ligne A décide d'ordonnancer τ_2 à l'instant 0 :**

- l'algorithme en-ligne A ordonnance la tâche τ_2 dès l'instant 0. À l'instant 1, il ordonnance la tâche τ_1 pour qu'elle respecte son échéance. À l'instant 2, une nouvelle tâche, la tâche τ_4, arrive dans le système :

$$\tau_4 : C_{4,1} = 4, X_4 = 3, C_{4,2} = 1, D_4 = 10$$

L'algorithme A ordonnance la tâche τ_4 dès l'instant 2 (pour respecter son échéance). À l'instant 6, la tâche τ_2 reprend son exécution après avoir fini sa suspension. Mais il reste entre les instants 9 et 10 une seule unité de temps à l'algorithme A pour terminer l'exécution des tâches τ_1 et τ_4, ce qui représente deux unités de temps. Par conséquent, l'une des deux échéances est manquée. Ainsi l'ordonnancement de la configuration I par l'algorithme A n'est pas faisable. Cet ordonnancement est représenté par la figure 6.6.a.

- L'algorithme hors-ligne optimal ordonnance à l'instant, la tâche τ_1, à l'instant 1, τ_2 et à l'instant 2, τ_4. À l'instant 6, la tâche τ_2 finit sa suspension et continue son exécution pour la terminer à l'instant 7. Finalement, la tâche τ_1 achève son exécution à l'instant 9 et la tâche τ_4 à l'instant 10. La figure 6.6.b résume l'ordonnancement de la configuration de tâches I par l'adversaire hors-ligne.

Ainsi, si l'algorithme en-ligne choisit d'ordonnancer la tâche à suspension à l'instant 0 alors il existe une configuration de tâches à suspension qui est non ordonnançable par

l'algorithme en-ligne mais il existe un ordonnancement faisable de cette configuration (par l'adversaire).

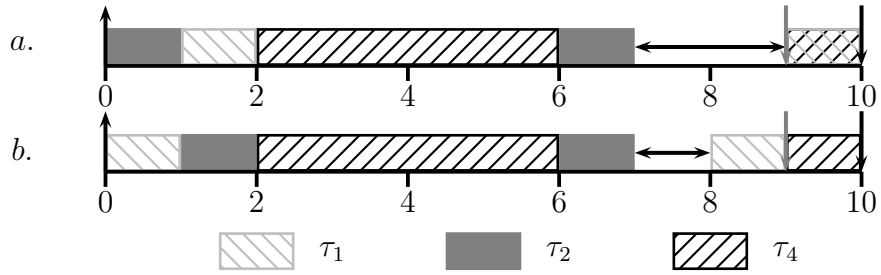

Figure 6.6. *The on-line algorithm A schedules τ_2 at time* 0

Pour conclure cette démonstration, quel que soit l'algorithme en-ligne considéré, en utilisant la configuration de tâches à suspension I, il est possible selon les décisions d'ordonnancement de l'algorithme en-ligne, de construire une configuration de tâches à suspension qui soit non ordonnançable par l'algorithme en-ligne, mais qui soit ordonnnançable par d'autres algorithmes, en particulier, l'adversaire hors-ligne.

□

6.5. Conclusion

Dans ce chapitre, plusieurs résultats négatifs préalables à une étude plus complète sur l'ordonnancement des tâches à suspension ont été démontrés. Des résultats négatifs concernant la complexité, la présence d'anomalies et l'optimalité des algorithmes en-ligne.

Le premier résultat prouvé est un résultat de complexité. Nous avons prouvé que l'ordonnancement de tâches qui peuvent se suspendre au plus une fois est un problème $\mathcal{NP}$-Difficile au sens fort. De plus nous avons établi qu'il n'existe pas d'algorithme universel pour l'ordonnancement de tâches à suspension sauf si $\mathcal{P} = \mathcal{NP}$. Ce résultat a fait l'objet de communications [RID 04, RID 05].

Nous avons établi la présence possible d'anomalies d'ordonnancement en exécutant des tâches à suspension, sous la politique d'ordonnancement EDF et également pour les algorithmes à priorité fixe (RM et DM). Ce résultat a été publié dans une conférence [RID 05].

Finalement, en utilisant l'analyse de compétitivité, nous avons démontré qu'aucun algorithme en-ligne déterministe n'est optimal pour l'ordonnancement de configurations de tâches à suspension sporadiques sur des systèmes monoprocesseurs. Ces résultats ont été publiés dans [RID 06a].

Tous ces résultats font également l'objet d'une revue internationale [RID 06b].

6.6. Bibliographie

[GAR 79] GAREY M., JOHNSON D., *Computers and Intractability : a guide to the theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[JEF 91] JEFFAY K., STANAT D., MARTEL C., « On Non-Preemptive Scheduling of Periodic and Sporadic tasks », *proc. Real-Time Systems Symposium*, p. 129-139, 1991.

[LIU 00] LIU J., *Real-Time Systems*, Prentice hall, 2000.

[RIC 03] RICHARD P., « On the complexity of scheduling real-time tasks with self-suspensions on one processor », *IEEE Euromicro Conf. on Real-Time Systems (ECRTS'03)*, page 8p, 2003.

[RID 04] RIDOUARD F., RICHARD P., COTTET F., « Negative results for scheduling independent hard real-time tasks with self-suspensions », 25^{th} *IEEE International Real-Time Systems Symposium (RTSS'04)*, 2004.

[RID 05] RIDOUARD F., RICHARD P., COTTET F., « Ordonnancement des tâches indépendantes avec suspension », *proc. Real-Time Embedded Systems (RTS'05), Paris*, 2005.

[RID 06a] RIDOUARD F., RICHARD P., « Worst-case analysis of feasibility tests for self-suspending tasks », *proc. International Conference on Real-Time and Network Systems (RTNS 2006), Poitiers*, vol. 1, n°30-31, p. 15–24, mai

[RID 06b] RIDOUARD F., RICHARD P., COTTET F., TRAORE K., « Some results on scheduling tasks with self-suspensions », *Journal of Embedded Computing (JEC), to appear*, 2006.

Chapitre 7

Compétitivité des algorithmes classiques d'ordonnancement

7.1. Introduction

Dans ce chapitre, nous étudions les algorithmes en-ligne classiques. Jusqu'à présent, aucune étude n'évalue la performance des algorithmes en-ligne pour l'ordonnancement de tâches à suspension. Cette recherche a pour but d'étudier les algorithmes en-ligne classiques : RM, DM, EDF et LLF (*cf.* paragraphe 4.2).

Cette étude cherche à évaluer la compétitivité des algorithmes en-ligne classiques selon deux critères de performance :

– la minimisation du nombre de tâches en retard ;
– la minimisation du temps de réponse maximum.

Nous évaluons les comportements des algorithmes en-ligne vis-à-vis des critères de performance, nous utilisons l'analyse de compétitivité. Nous démontrons dans ce chapitre que les algorithmes en-ligne classiques ne sont pas ou peu compétitifs pour chacun de ces deux critères de performance même avec des configurations de tâches à suspension ayant un facteur d'utilisation faible. Pour simplifier les résultats, nous supposons que les périodes des tâches sont prises suffisamment grandes pour qu'une unique occurrence de chaque tâche appartienne à l'hyperpériode.

Ce chapitre est subdivisé en deux paragraphes. Le paragraphe 7.2 étudie la compétitivité des algorithmes en-ligne classiques pour la minimisation du nombre de tâches en retard et le paragraphe 7.3 traite la compétitivité des algorithmes en-ligne classiques pour la minimisation du temps de réponse maximum.

7.2. Minimisation du nombre de tâches en retard

Le critère de minimisation du nombre de tâches en retard est développé dans cette partie. Ce critère est bien sûr équivalent à celui de la maximisation du nombre de tâches respectant leurs échéances. Nous allons rappeler tout d'abord les principaux résultats connus puis nous étudierons les performances d'EDF et LLF pour l'ordonnancement de tâches à suspension. À

noter que dans la suite, nous considèrons que les tâches doivent toujours finir leur exécution même si ces dernières sont en retard.

Nous savons qu'il n'existe pas d'algorithme en-ligne compétitif pour la maximisation du nombre de tâches respectant leurs échéances en ordonnançant des systèmes de tâches classiques, sans suspension. Mais il en existe dans des cas particuliers comme présentés dans [BAR 01, BAR 94]. Dans ce contexte particulier, nous allons prouver qu'en ordonnançant des tâches à suspension, les algorithmes classiques comme EDF ne sont pas compétitifs. Il est à noter que nos résultats sont valides d'un point de vue de la faisabilité puisque les résultats que nous allons présenter, le sont avec pour les configurations étudiées, des facteurs d'utilisation faibles, proches de zéro.

7.2.1. *Résultats connus*

BARUAH *et al.* [BAR 01, BAR 94] ont prouvé qu'il n'existe pas d'algorithme d'ordonnancement en-ligne préemptif pour maximiser le nombre de tâches respectant leurs échéances dans les systèmes monoprocesseurs. Pour obtenir ce résultat, l'adversaire définit une configuration de tâches avec des surcharges du processeur. Mais ces auteurs montrent qu'il existe des résultats positifs dans des cas particuliers. Nous allons maintenant présenter l'un de ces cas.

La définition 20 présente les propriétés de l'un de ces cas particuliers.

Définition 20 *Monotonic Absolute Deadlines (MAD) :*
Un système de tâches est qualifié de MAD si l'échéance absolue de chaque nouvelle tâche qui arrive dans le système est supérieure ou égale à celle de n'importe quelle tâche déjà apparue dans le système.

La définition 21 présente l'algorithme *SRPTF*.

Définition 21 *Shortest Remaining Processing Time First (SRPTF) :*
SRPTF est un algorithme d'ordonnancement en-ligne préemptif. Il alloue à chaque instant, le processeur à la tâche ayant le plus petit temps processeur restant à exécuter.

Lorsque *SRPTF* est utilisé pour ordonnancer des tâches à suspension, deux définitions sont possibles suivant que la durée de suspension est incluse ou non dans le temps processeur :

– si on considère que la suspension est incluse dans le temps processeur alors *SRPTF* ordonnance à l'instant t, la tâche ayant la plus petite valeur $c_i(t) + x_i(t)$ où $c_i(t)$ (respectivement $x_i(t)$) désigne le temps processeur restant à ordonnancer (respectivement le délai de suspension) à l'instant t.

– sinon, l'algorithme *SRPTF* ordonnance la tâche avec la plus petite valeur de $c_i(t)$.

Dans la suite, nous ne considérerons exclusivement que la deuxième définition de l'algorithme en-ligne *SRPTF*.

Dans [BAR 01, BAR 94], BARUAH *et al.* ont prouvé que si un système de tâches a la propriété *MAD* alors l'algorithme d'ordonnancement en-ligne *SRPTF* est 2-compétitif pour minimiser le nombre de tâches en retard. De plus, ils ont démontré que la borne inférieure pour ce problème est également 2, c'est-à-dire que la borne est serrée. Par conséquent, *SRPTF* est donc l'un des meilleurs algorithmes en-ligne possibles.

7.2.2. *Mauvais résultats pour l'ordonnancement de tâches à suspension*

Dans ce paragraphe, nous démontrons qu'en ordonnançant des systèmes de tâches à suspension, l'algorithme d'ordonnancement *SRPTF* n'est plus compétitif pour minimiser le nombre de tâches en retard et ce même si le système est *MAD* :

Theorème 9 *Pour les systèmes de tâches à suspension MAD même avec un facteur d'utilisation arbitrairement petit, l'algorithme en-ligne SRPTF n'est pas compétitif pour maximiser le nombre de tâches respectant leurs échéances.*

Démonstration :

La démonstration se fait en utilisant l'analyse de compétitivité avec un adversaire évolutif. L'adversaire génère la configuration I à $n+1$ tâches suivante (où n est un entier positif) :

$$\tau_0 : r_0 = 0, C_{01} = 1, X_1 = 0, C_{02} = 0, D_0 = K$$
$$\tau_i : r_i = i-1, C_{i1} = 1, X_i = K-2, C_{i2} = 1, D_i = K \quad avec \quad i \in \{1, .., n\}$$

Où K est un entier positif strictement supérieur à 1.

La configuration de tâches à suspension I respecte bien la propriété *MAD*.

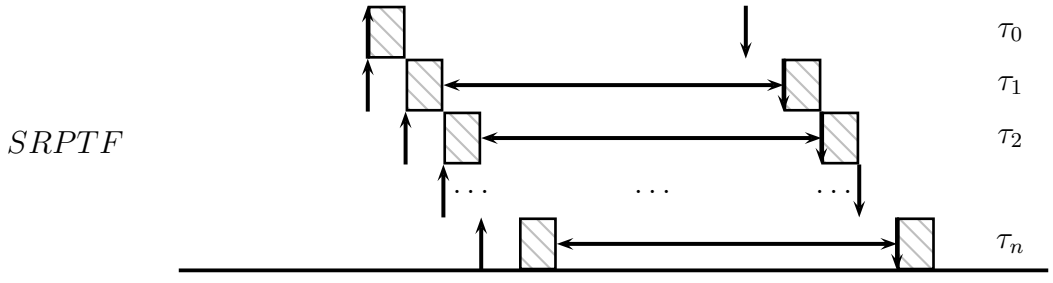

Figure 7.1. *Ordonnancement de la configuration I par SRPTF pour la minimisation du nombre de tâches en retard*

La figure 7.1 présente l'ordonnancement de I par l'algorithme d'ordonnancement *SRPTF* et par son adversaire clairvoyant. À la date 0, τ_0 et τ_1 sont disponibles, *SRPTF* ordonnance τ_0

parce que τ_0 a le plus petit temps processeur restant. Ensuite, pour chaque tâche τ_i, ($1 \leq i \leq n$), la première sous-tâche est ordonnancée à la date i et la seconde à la date $K + (i - 1)$.

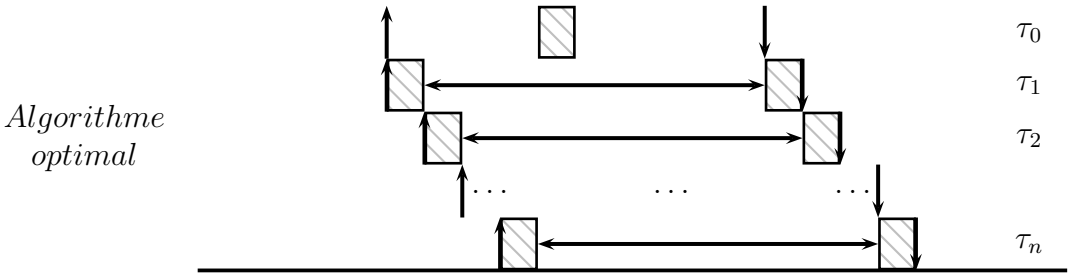

Figure 7.2. *Ordonnancement de la configuration I par l'adversaire optimal pour la minimisation du nombre de tâches en retard*

La figure 7.2 présente l'ordonnancement de I par l'adversaire clairvoyant. L'adversaire choisit d'ordonnancer la tâche τ_1 à l'instant 0. Puis, il ordonnance chaque tâche τ_i ($i \in \{2, \ldots, n\}$) à l'instant $i - 1$. Pour finir, il ordonnance τ_0.

Calcul du ratio de compétitivité de l'algorithme *SRPTF* :

$$c_{SRPTF} = \frac{\sigma_{SRPTF}}{\sigma^*} = \lim_{n \to \infty} \frac{1}{n+1} = 0$$

Et pour finir, calculons le facteur d'utilisation de I :

$$\begin{aligned} U_I &= \sum_{i=0}^{n} \frac{C_{i,1}+C_{i,2}}{T_i} = \frac{1}{K-1} + \sum_{i=1}^{n} \frac{2}{K} \\ &= \lim_{K \to \infty} \frac{2n+1}{K} = 0 \end{aligned}$$

Pour conclure, une configuration de tâches à suspension I a été générée par l'adversaire. Elle respecte la propriété du *MAD* et a un faible facteur d'utilisation alors que *SRPTF* n'est pas compétitif pour maximiser le nombre de tâches respectant leurs échéances.

□

Avec la même configuration I que celle présentée dans la démonstration du théorème 9, les résultats obtenus pour *SRPTF* peuvent s'étendre à EDF, DM and *RM*.

Corollaire 2 *Pour des systèmes de tâches à suspension et ayant la propriété du MAD,* EDF, DM *et* RM *ne sont pas compétitifs pour maximiser le nombre de tâches respectant leurs échéances.*

Démonstration :

Soit I la configuration générée par l'adversaire lors de la démonstration du théorème 9. Sur cette configuration, EDF, DM et RM assignent exactement les mêmes priorités aux tâches que *SRPTF* le fait. Par conséquent, nous obtenons les mêmes ratios de compétitivité et facteurs d'utilisation pour ces algorithmes que pour *SRPTF*. Finalement, la même conclusion s'impose à savoir que ni EDF, ni DM et ni RM ne sont compétitifs pour maximiser le nombre de tâches respectant leurs échéances.

□

Nous nous intéressons maintenant à l'étude de l'algorithme LLF. Comme nous l'avons vu dans le paragraphe 4.2, cet algorithme affecte la plus haute priorité à la tâche ayant la plus petite laxité dynamique. Mais lorsque LLF est utilisé pour les tâches à suspension, et suivant que nous considérons ou non le délai de suspension de la tâche comme étant de la laxité de la tâche, deux définitions de l'algorithme LLF peuvent être données :

– si le délai de suspension des tâches est considéré comme laxité de la tâche, la laxité dynamique L_1 de la tâche τ_i est égale à $L_{i,1}(t) = d_i - t - c_i(t)$; où $c_i(t)$ est le temps processeur de la tâche τ_i restant à exécuter à l'instant t ;

– sinon, la laxité dynamique L_2 de la tâche τ_i est égale à $L_{i,2}(t) = d_i - t - c_i(t) - x_i(t)$; où $x_i(t)$ est le délai de suspension de la tâche τ_i restant à exécuter à l'instant t.

Dans tout ce qui suit, nous ne considérons que la première définition de la laxité dynamique (L_1). Ainsi, la durée de suspension de la tâche est comprise dans le calcul de sa laxité.

Theorème 10 *Pour des systèmes de tâches à suspension et ayant la propriété MAD,* LLF *n'est pas compétitif pour maximiser le nombre de tâches respectant leurs échéances.*

Démonstration :

Pour l'algorithme *L*east *L*axity *F*irst (LLF), soit I la configuration à n tâches suivante :

$$\tau_i : r_i = 0, C_{i1} = 3, X_i = K - 3(n+1), C_{i2} = 3, D_i = K \quad avec \quad \forall i = 1..n$$

où K est un nombre très grand tendant vers l'infini.

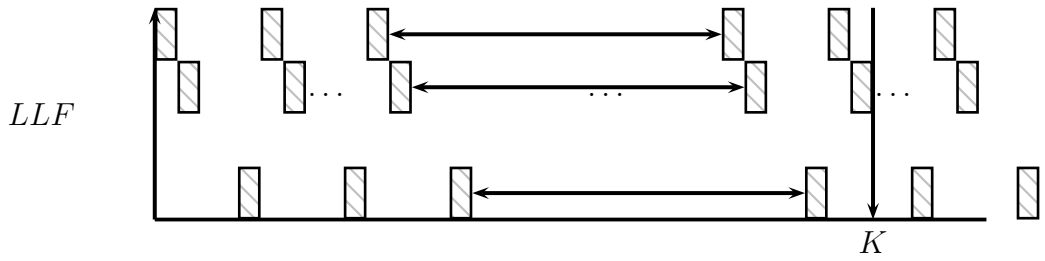

Figure 7.3. *Ordonnancement de la configuration I par* LLF *pour la minimisation du nombre de tâches en retard*

Les résultats de l'ordonnancement de I par LLF sont donnés par la figure 7.3. À l'instant 0, LLF commence par ordonnancer la tâche τ_1. Mais après avoir exécuté τ_1 pendant une unité de temps, les autres tâches τ_i ($i \in 2, .., n$) ont une priorité plus importante que celle de τ_1. Par conséquent, LLF préempte τ_1 et exécute τ_2. Mais, après une unité de temps d'exécution de τ_2, les autres tâches ont une priorité plus importante que celle de τ_2. Les tâches τ_i ($i \in 1, .., n$) ont la même laxité obligeant LLF à préempter à chaque fois la tâche active après seulement une unité de temps de son exécution. En conséquence, l'exécution des tâches ne peut se faire qu'unité après unité. Ce qui oblige LLF à ne respecter aucune des échéances de ces tâches.

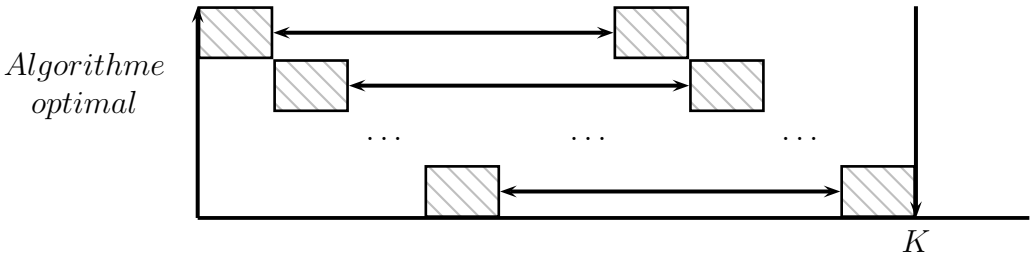

Figure 7.4. *Ordonnancement de la configuration I par un algorithme optimal pour la minimisation du nombre de tâches en retard*

L'ordonnancement de la configuration I par l'adversaire clairvoyant est présenté par la figure 7.4. L'algorithme optimal ordonnance la première sous-tâche de τ_1 puis la première sous-tâche de τ_2 pour finir avec la première sous-tâche de τ_n et dans le même ordre, la seconde sous-tâche de ces tâches. Par conséquent, l'algorithme optimal ne manque aucune échéance.

Le calcul du ratio de compétitivité de LLF donne :

$$c_{LLF} = \frac{\sigma_{LLF}}{\sigma^*} = \frac{0}{n} = 0$$

Ce qui implique que LLF n'est pas compétitif pour la maximisation du nombre de tâches respectant leurs échéances. Et le calcul du facteur d'utilisation de cette configuration I (cf. 7.1), montre bien qu'il est quasi nul. Ce qui nous permet de conclure quant à la faiblesse de LLF.

$$U_I = \sum_{i=1}^{n} \frac{C_{i1} + C_{i2}}{T_i} = \lim_{K \to \infty} \frac{6n}{K} = 0 \tag{7.1}$$

□

7.2.3. *Technique de l'augmentation de ressources*

En analyse de compétitivité, l'algorithme en-ligne et l'adversaire clairvoyant s'exécutent sur la même machine. Mais les résultats obtenus par cette technique d'analyse sont quelques fois pessimistes. Afin de diminuer ce pessimisme, des alternatives existent. De manière à analyser les performances relatives des algorithmes sous un autre angle, une extension de l'analyse de compétitivité a été créée : l'augmentation de ressources [PHI 97]. Dans cette extension, l'algorithme en-ligne s'exécute sur une machine plus rapide que celle de l'adversaire (il est donc avantagé...). Un résultat d'ordonnancement connu est que l'algorithme d'ordonnancement EDF n'est pas optimal pour la minimisation du nombre de tâches manquant leur échéance (ou fautes temporelles) en présence de surcharge du processeur (*cf.* paragraphe 4.3.2). Il a été prouvé dans [PHI 97], qu'en présence de surcharge, si une configuration de tâches qui est, avec une même machine, non ordonnançable par EDF mais ordonnançable par un algorithme optimal, alors elle sera ordonnançable par EDF avec une machine deux fois plus rapide. Cette dernière assertion est fausse dès que le système est composé de tâches à suspension et nous le démontrons.

Le résultat connu d'EDF ne tient plus si les tâches sont à suspension. Le théorème qui suit, va démontrer que même s'il existe une configuration I ordonnançable par un algorithme optimal, alors elle ne sera pas toujours ordonnançable par EDF même sur une machine k fois plus rapide, où k est un entier strictement positif quelconque.

Pour ordonnancer des tâches à suspension, le système est composé de plusieurs processeurs : le processeur principal pour ordonnancer les tâches proprement dites et des processeurs dédiés (ou annexes) pour l'exécution des opérations externes. Par conséquent, il existe deux types de processeurs qui peuvent être accélérés. Dans un premier temps, nous accélérons uniquement le processeur principal et dans un deuxième temps, uniquement les processeurs annexes.

7.2.3.1. *Technique d'augmentation de ressources : le processeur principal*

Nous prouvons dans ce paragraphe, que pour l'ordonnancement de tâches à suspension, l'algorithme EDF ne peut pas toujours définir un ordonnancement faisable avec un processeur de vitesse s (où s est un entier quelconque strictement positif) alors qu'il existe un ordonnancement faisable par un algorithme hors-ligne sur une machine de vitesse 1 (déterminé par un algorithme optimal et clairvoyant). Par conséquent, allouer plus de ressources à l'algorithme EDF n'aide pas à définir un simple algorithme en-ligne compétitif.

Theorème 11 *Augmenter la vitesse s (où s est un entier quelconque strictement positif) du processeur n'améliore pas les performances d'*EDF *quand les tâches sont autorisées à se suspendre par rapport à un algorithme d'ordonnancement optimal utilisant un processeur de vitesse unitaire.*

Démonstration :

Raisonnement par l'absurde :

Hypothèse : Il existe un entier s, $s > 1$ tel que si EDF dispose d'une machine s fois plus rapide que celle de son adversaire clairvoyant, alors EDF est optimal.

Soit la configuration I à $n+1$ tâches suivante :

$$\tau_0 \ : \ r_0 = 0,\ C_{0,1} = 2s,\ X_0 = 0,\ C_{0,2} = 0,\ D_0 = 4s+1$$
$$\tau_i \ : \ r_i = 0,\ C_{i,1} = 1/n,\ X_i = 4s,\ C_{i,2} = 1/n,\ D_i = 4s+2 \qquad 1 \leq i \leq n$$

– La figure 7.5.a donne l'ordonnancement de I par un algorithme optimal sur une machine de vitesse 1. L'algorithme optimal ordonnance dans un premier temps, successivement, toutes les tâches τ_i avec $1 \leq i \leq n$, puis, pendant la suspension de celles-ci, il ordonnance la tâche τ_0. Ainsi, toutes les échéances sont respectées.

– La figure 7.5.b présente l'ordonnancement de I par l'algorithme EDF sur une machine s fois plus rapide. A l'instant 0, EDF ordonnance τ_0 car son échéance est plus proche que celle des autres tâches. Ainsi, il retarde l'exécution des tâches τ_i ($1 \leq i \leq n$) qui manquent leur échéance. Par conséquent, EDF même avec une machine s fois plus rapide ne parvient toujours pas à respecter toutes les échéances et par conséquent à construire un ordonnancement faisable.

Calculons maintenant le ratio de compétitivité de l'algorithme d'ordonnancement EDF avec une machine s fois plus rapide. L'algorithme optimal respecte quant à lui toutes les échéances comme le montre la figure 7.5.a à la différence d'EDF qui ne respecte qu'une seule échéance, celle de τ_1 (*cf.* figure 7.5.b). Notons par σ_{EDF}, la performance obtenue par l'algorithme EDF, et par σ^*, celle de l'algorithme optimal. Le ratio de compétitivité d'EDF en faisant tendre le nombre de tâches, n, vers l'infini est égal à :

$$c_{\text{EDF}} = \frac{\sigma_{\text{EDF}}}{\sigma^*} = \lim_{n \to \infty} \frac{1}{n+1} = 0$$

EDF n'est par conséquent pas compétitif pour la minimisation du nombre de tâches ne respectant pas leur échéance, même en lui augmentant la vitesse de sa machine allouée.

Ce qui contredit l'hypothèse de départ et conclut cette démonstration.

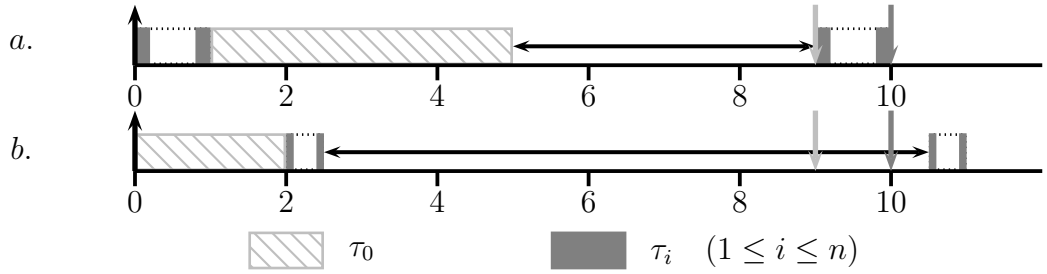

Figure 7.5. *L'ordonnancement de I par* EDF *avec un processeur de vitesse* 2 *($s = 2$)*

□

7.2.3.2. *Technique d'augmentation de ressources : les processeurs annexes*

Nous démontrons dans ce paragraphe que l'augmentation de la vitesse des processeurs dédiés n'améliore pas les performances de l'algorithme EDF. La vitesse du processeur principal reste unitaire.

Theorème 12 *L'augmentation de la vitesse des processeurs annexes pour l'algorithme d'ordonnancement en-ligne* EDF *n'améliore pas ses performances quand les tâches sont allouées pour se suspendre au plus une fois.*

Démonstration :

Pour démontrer ce théorème, nous employons la même méthode utilisée lors de la démonstration du théorème 11 : une démonstration par l'absurde.

Hypothèse : il existe un entier strictement positif s, $s > 1$ tel que l'algorithme EDF avec des processeurs annexes de vitesse s fois plus rapides que ceux de l'algorithme optimal alors, il est optimal. Par conséquent, s'il existe un ordonnancement faisable pour une configuration I alors il existe un ordonnancement faisable de I par EDF avec des processeurs annexes de vitesse s.

Soit I la configuration suivante :

$$\tau_0 : r_0 = 0, C_{0,1} = 2s, X_0 = 0, C_{0,2} = 0, D_0 = 2s + 1$$
$$\tau_i : r_i = 0, C_{i,1} = 1/n, X_i = 2s, C_{i,2} = 1/n, D_i = 2s + 2 \qquad 1 \leq i \leq n$$

– L'ordonnancement de la configuration I par l'algorithme optimal est représenté par la figure 7.6.a. À l'instant 0, toutes les tâches sont activées, l'algorithme optimal clairvoyant avec des processeurs dédiés de vitesse 1 ordonnance les tâches τ_i à l'instant 0 (avec $1 \leq i \leq n$). Enfin, il ordonnance la tâche τ_0 pendant la suspension des tâches τ_i ($1 \leq i \leq n$). Toutes les échéances sont respectées.

– La figure 7.6.b illustre l'ordonnancement de I par EDF avec des processeurs dédiés de vitesse s. À l'instant 0, EDF ordonnance τ_0 en premier car son échéance est la plus proche. Ensuite, il ordonnance les tâches τ_i ($1 \leq i \leq n$). Par conséquent, toutes les tâches τ_i ($1 \leq i \leq n$) manquent leur échéance.

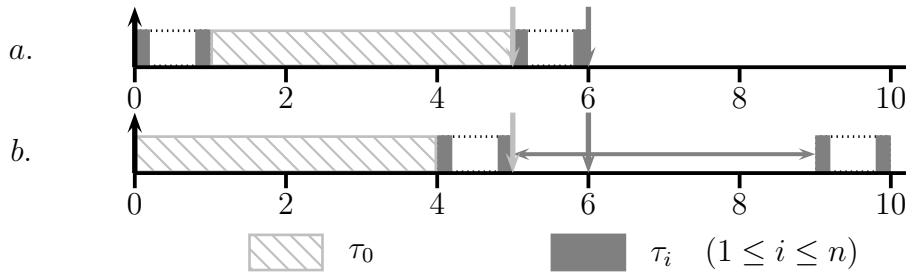

Figure 7.6. *L'ordonnancement de I par* EDF *avec des processeurs dédiés de vitesse 2 ($s = 2$)*

□

Par conséquent, l'algorithme en-ligne EDF même avec des processeurs dédiés s fois plus rapides que ceux de l'algorithme optimal ne peut obtenir un ordonnancement faisable de I alors qu'il existe un ordonnancement faisable de cette configuration avec des processeurs dédiés de vitesse unitaire.

Pour clore cette démonstration, calculons le ratio de compétitivité (pour la minimisation du nombre de tâches en retard) de l'algorithme EDF lorsqu'il dispose de processeurs dédiés s fois plus rapides que ceux de l'adversaire.

– L'algorithme optimal respecte toutes les échéances des tâches, donc, si nous notons σ^* sa performance, $\sigma^* = n + 1$.

– L'algorithme EDF, ne respecte que l'échéance de la tâche τ_0. En conséquence, si nous notons σ_{EDF} sa performance, $\sigma_{EDF} = 1$.

Le ratio de compétitivité résultant (en faisant tendre le nombre de tâche n vers l'infini) est de :

$$c_{EDF} = \frac{\sigma_{\text{EDF}}}{\sigma^*} = \lim_{n \to \infty} \frac{1}{n+1} = 0$$

La technique d'augmentation de ressources utilisée ainsi, est inefficace pour EDF. Augmenter seulement le processeur principal ou seulement les processeurs annexes n'améliore pas les performances de l'algorithme d'ordonnancement EDF. Une issue intéressante serait d'étudier la technique d'augmentation de ressources appliquée à tout le système (processeur principal et processeurs dédiés).

7.3. Minimisation du temps de réponse maximum

Lorsque les tâches ne se suspendent pas, pour tout algorithme d'ordonnancement conservatif (c.-à-d. n'insérant pas de temps creux dans l'ordonnancement s'il existe au moins une tâche prête à s'exécuter), alors le plus grand temps de réponse d'une tâche ne peut excéder la durée de la période d'activité synchrone (*Synchronous Busy Period*). Mais cette assertion devient inexacte quand les tâches sont à suspension comme l'illustrent les résultats suivants pour RM, DM, EDF et LLF.

7.3.1. *Compétitivité d'*EDF, RM *et* DM

La politique d'ordonnancement EDF est au mieux 2-compétitive pour minimiser le temps de réponse maximum.

Theorème 13 *L'algorithme d'ordonnancement* EDF *est au mieux* 2*-compétitif pour la minimisation du temps de réponse maximum en ordonnançant des tâches à suspension, se suspendant au plus une fois.*

Démonstration :

Soit la configuration de tâches I suivante :

$$\tau_1 : r_1 = 0, C_{1,1} = \epsilon, X_1 = K, C_{1,2} = \epsilon, D_1 = 4K$$
$$\tau_2 : r_2 = 0, C_{2,1} = K, X_2 = 0, C_{2,2} = 0, D_2 = 4K - 1$$

Où K est un entier positif supérieur à 1.
Où ϵ est un nombre positif strictement inférieur à 1 et tendant vers zéro. Sur la figure 7.7 sont présentés les résultats de l'ordonnancement de la configuration I par l'algorithme d'ordonnancement EDF et par un algorithme clairvoyant optimal. L'algorithme EDF ordonnance en premier la tâche τ_2 car son échéance est la plus proche. Mais l'algorithme clairvoyant, ordonnance en premier la tâche τ_1 pour permettre d'ordonnancer τ_2 pendant la suspension de τ_1.

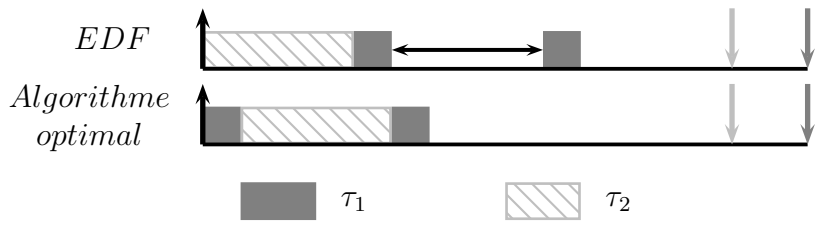

Figure 7.7. *Compétitivité d'*EDF *pour la minimisation du temps de réponse maximum*

Le calcul du ratio de compétitivité d'EDF finit cette démonstration. Le temps de réponse obtenu par EDF pour l'ordonnancement de I est égal à $2K+2\epsilon$ alors que celui de l'algorithme optimal est de $K + 2\epsilon$. Ce qui nous permet d'obtenir un ratio de compétitivité d'EDF de 2 en faisant tendre ϵ vers 0 (*cf.* figure 7.2).

$$c_{EDF} = \frac{\sigma_{EDF}}{\sigma^*} = \lim_{\epsilon \to 0} \frac{2K + 2\epsilon}{K + 2\epsilon} = \frac{2K}{K} = 2 \tag{7.2}$$

Ce qui conclut notre démonstration puisqu'une configuration I a été trouvée sur laquelle le ratio de compétitivité de l'algorithme d'ordonnancement EDF est de 2. Par conséquent, EDF est un algorithme au mieux 2-compétitif.

□

Nous allons maintenant étendre ce résultat aux algorithmes RM et DM.

Corollaire 3 *Les algorithmes d'ordonnancement* RM *et* DM *sont au mieux* 2*-compétitifs pour l'ordonnancement de tâches à suspension où chaque tâche se suspend au plus une seule fois, et en minimisant le temps de réponse maximum.*

Démonstration :

La configuration I utilisée dans la démonstration du théorème 13 est conservée. Si nous ordonnançons cette configuration I avec les algorithmes RM et DM, les priorités affectées par ces algorithmes seront les mêmes que celles affectées par EDF. Par conséquent, comme EDF dans la démonstration du théorème 13, RM et DM ordonnancent τ_2 avant τ_1, ce qui implique que les mêmes résultats que pour l'algorithme EDF sont obtenus. Ainsi, RM et DM ont un ratio de compétitivité d'au moins deux. Ce qui permet de conclure que RM et DM sont au mieux 2-compétitifs.

□

7.3.2. *Compétitivité de* LLF

La politique d'ordonnancement LLF est au mieux 2-compétitive pour minimiser le temps de réponse.

Theorème 14 *L'algorithme d'ordonnancement* LLF *est au mieux* 2*-compétitif pour la minimisation du temps de réponse maximum pour l'ordonnancement de tâches à suspension et se suspendant au plus une fois.*

Démonstration :

Pour démontrer ce théorème, le ratio de compétitivité de LLF en ordonnançant une certaine configuration I doit être égal à 2. Soit I la configuration suivante :

$$\tau_1 : r_1 = 0, C_{1,1} = \epsilon, X_1 = K, C_{1,2} = \epsilon, D_1 = 4K$$
$$\tau_2 : r_2 = 0, C_{2,1} = K, X_2 = 0, C_{2,2} = 0, D_2 = 2K + 2$$

Où K est un entier positif supérieur à 1.
Où ϵ est un nombre positif strictement inférieur à 1 et tendant vers zéro. La figure 7.8 montre les résultats de l'ordonnancement de la configuration de tâches I par l'algorithme d'ordonnancement LLF et par un algorithme clairvoyant optimal. L'algorithme LLF (*cf.* figure 7.8) ordonnance en premier la tâche τ_2 car sa laxité dynamique est plus faible. Quant à l'algorithme optimal, il commence à ordonnancer τ_1 pour permettre d'exécuter τ_2 pendant la suspension de τ_1.

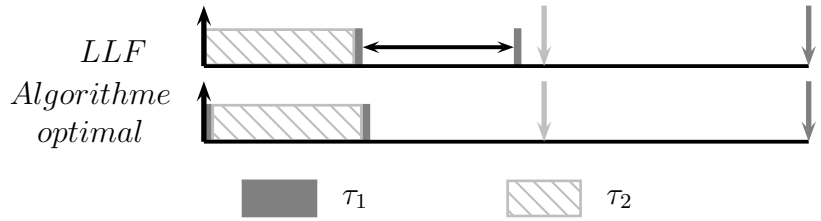

Figure 7.8. *Compétitivité de* LLF *pour la minimisation du plus grand temps de réponse*

Enfin, calculons le ratio de compétitivité de LLF (7.3), en sachant que la longueur d'ordonnancement obtenue par LLF est égale à $2K - 2\epsilon$ et que celle obtenue par un algorithme optimal n'est que de $K + 2\epsilon$.

$$c_{LLF} = \frac{\sigma_{LLF}}{\sigma^*} = \lim_{\epsilon \to 0} \frac{2K - 2\epsilon}{K + \epsilon} = \frac{2K}{K} = 2 \tag{7.3}$$

Ce qui permet de conclure la démonstration puisqu'une configuration I sur laquelle le ratio de compétitivité de l'algorithme d'ordonnancement LLF est de 2 a été déterminée. Et donc, LLF est un algorithme au mieux 2-compétitif.

□

7.4. Conclusion

Dans ce chapitre, l'analyse de compétitivité a été utilisée pour évaluer les performances des algorithmes en-ligne classiques (*RM*, DM, EDF et LLF paragraphe 4.2) dans l'étude de deux critères de performance : la minimisation du nombre de tâches en retard et la minimisation du temps de réponse maximum.

Ainsi, en utilisant l'analyse de compétitivité, nous avons démontré que les algorithmes classiques en-ligne sont non compétitifs pour minimiser le nombre de tâches en retard. En particulier, ils sont même non compétitifs sur des configurations de tâches à suspension avec des charges processeurs presque nulles alors qu'il existe des ordonnancements faisables de ces configurations générées. Pour ce même problème, nous avons également démontré que la technique d'augmentation des ressources n'amène aucune amélioration pour l'ordonnancement de tâches à suspension sous EDF. Ces résultats ont été publiés dans des conférences [RID 04, RID 05], et une revue [RID 06].

Nous avons finalement démontré que les algorithmes en-ligne classiques ne sont pas mieux que 2-compétitifs pour la minimisation du temps de réponse maximum. Ces résultats ont fait l'objet d'une communication [RID 05], et d'une revue [RID 06].

7.5. Bibliographie

[BAR 94] BARUAH S., HARITSA J., SHARMA N., « On-line scheduling to maximise task completions », *in : proc. Real-Time Systems Symposium*, p. 228-236, 1994.

[BAR 01] BARUAH S., HARITSA J., SHARMA N., « On-line scheduling to maximise task completions », *The Journal of Combinatorial Mathematics and Combinatorial Computing*, vol. 39, p. 65-78, 2001.

[PHI 97] PHILIPS C., STEIN C., TORNG E., WEIN J., « Optimal time-critical scheduling via resource augmentation », *prc. 29^{th} Ann. ACM Symp. on Theory of Computing*, p. 110–149, 1997.

[RID 04] RIDOUARD F., RICHARD P., COTTET F., « Negative results for scheduling independent hard real-time tasks with self-suspensions », *25^{th} IEEE International Real-Time Systems Symposium (RTSS'04)*, 2004.

[RID 05] RIDOUARD F., RICHARD P., COTTET F., « Ordonnancement des tâches indépendantes avec suspension », *proc. Real-Time Embedded Systems (RTS'05), Paris*, 2005.

[RID 06] RIDOUARD F., RICHARD P., COTTET F., TRAORE K., « Some results on scheduling tasks with self-suspensions », *Journal of Embedded Computing (JEC), to appear*, 2006.

Chapitre 8

Analyse des tests d'ordonnançabilité pour les systèmes à priorité fixe

8.1. Introduction

Plusieurs tests d'ordonnançabilité ont été présentés (*cf.* chapitre 5) pour l'ordonnancement monoprocesseur de tâches qui peuvent se suspendre au plus une seule fois. Ces tests sont basés sur les algorithmes à priorité fixe (RM et DM). Ces tests d'ordonnançabilité utilisent une des deux méthodes suivantes :

– le calcul du pire temps de réponse : pour chaque tâche, le pire temps de réponse est calculé et la configuration est ordonnançable si le pire temps de réponse de chaque tâche est inférieur à son échéance relative ;

– le calcul de la charge processeur : la charge du processeur est calculée et ne doit pas dépasser 1 (qui représente les capacités d'ordonnancement du processeur) pour que la configuration soit ordonnançable.

Définir une méthode d'ordonnancement en-ligne revient à choisir d'une part, un algorithme d'ordonnancement, et d'autre part, un test d'ordonnançabilité. Nous avons vu dans le chapitre 6 qu'aucun algorithme en-ligne ne peut être optimal pour ordonnancer des tâches à suspension. En conséquence, choisir un ordonnanceur à priorité fixe va introduire du pessimisme dans la méthode d'ordonnancement. Mais du pessimisme peut aussi être introduit dans la méthode à travers le test d'ordonnançabilité.

L'objectif de ce chapitre est d'évaluer les tests d'ordonnançabilité. En effet, les tests présentés dans le chapitre 5 sont des tests approchés. Ainsi, ils induisent nécessairement du pessimisme dans leurs calculs. Ces méthodes sont connues, même utilisées dans certains cas, mais aucune évaluation de leur pessimisme n'est connue. De plus, personne n'a cherché à savoir si un test, et donc sa manière de prendre en compte les suspensions des tâches, est fondamentalement meilleur que les autres. Ce chapitre mène une première étude sur ces tests, mais pour qu'elle soit pertinente, les tests à étudier doivent avoir des critères communs : le même procédé de calculs et basés sur le même algorithme. Seulement deux tests d'ordonnançabilité

sont basés sur EDF. De plus, ils sont basés sur deux calculs différents. Ainsi, notre choix se porte uniquement sur les algorithmes à priorité fixe.

Pour les algorithmes à priorité fixe, seuls des tests fondés sur le calcul du pire temps de réponse ont été établis. Notre étude porte sur les algorithmes à priorité fixe et le calcul du pire temps de réponse.

Dans ce chapitre, les tests d'ordonnançabilité s'appliquant aux tâches à suspension étudiés sont ceux de KIM *et al.* [KIM 95] et celui de LIU [LIU 00]. Le test de PALENCIA et GONZALEZ HARBOUR n'est pas abordé ici, car son implémentation et l'ajout de caractéristiques aux tâches (gigue sur activation, offset,...) rend sa mise en œuvre et son étude très difficile. Nous avons choisi de ne pas l'implémenter afin d'obtenir plus rapidement des résultats.

Pour l'analyse des tests d'ordonnançabilité, nous subdivisons ce chapitre en deux parties : une partie par méthode d'analyse. La première partie se concentre sur l'étude des tests d'ordonnançabilité par l'analyse de compétitivité qui permet d'obtenir une mesure quantitative du pessimisme introduit par le test dans le pire des cas à travers le ratio de compétitivité. Puis dans un second temps, nous utiliserons la simulation pour obtenir des comportements moyens de ces tests. Pour déterminer le pessimisme de ces tests d'ordonnançabilité, nous utilisons l'approche présentée par EPSTEIN et ROB VAN STEE dans l'article [EPS 01] et VESTJENS dans l'article [CHE 96, HOO 96]. Leurs travaux consistent à étudier les problèmes d'ordonnancement d'une façon originale, différente des autres études. En effet, leur approche consiste à étudier un problème d'ordonnancement en utilisant l'analyse de compétitivité. Ils ne cherchent pas à établir un nouvel algorithme qui soit optimal pour le problème étudié, ou à chercher une autre solution d'ordonnancement, mais à établir, pour une classe d'algorithmes donnée (par exemple, les algorithmes en-ligne et déterministes), une borne inférieure du ratio de compétitivité de tous les algorithmes de la classe. Ils ont ainsi déterminé ou amélioré la borne inférieure connue du ratio de compétitivité pour les algorithmes en-ligne déterministes (et randomisés) pour plusieurs critères de performance entre autres, la minimisation de la date de fin d'exécution, du temps de réponse moyen. Afin d'améliorer une borne inférieure, cette méthode consiste en la génération *brute force* de configuration de tâches, pour ne retenir au final que la configuration de tâches menant à la borne inférieure la plus importante, la nouvelle valeur prenant la place de borne inférieure.

Ainsi, pour étudier les tests d'ordonnançabilité, nous abordons le problème en terme d'analyse de compétitivité. Chacun des tests traités est basé sur le calcul du pire temps de réponse. Afin de calculer le ratio de compétitivité, nous devons disposer d'une méthode exacte du calcul du pire temps de réponse des tâches ordonnancées sous RM. Afin de calculer le temps de réponse maximum exact obtenu avec l'algorithme RM, nous utilisons le principe des travaux de [ALZ 05] qui ont défini une méthode exacte pour le calcul du temps de réponse de tâches dans les systèmes non préemptifs et lorsque les durées d'exécution des tâches varient. Nous avons adapté leur méthode aux tâches préemptives avec suspension, ordonnancées par RM. Pour déterminer la borne inférieure de chaque test, comme dans les travaux précédemment cités, nous générons bon nombre de configurations de tâches à suspension, nous calculons le ratio de compétitivité de chaque tâche et ainsi de chaque configuration. Finalement, nous ne conservons pour chaque test que la configuration menant au pire ratio de compétitivité.

La question importante que nous allons étudier dans ce chapitre est de savoir si ces trois tests sont d'une part, comparables (dominance d'un test par rapport aux autres) et d'autre part, quel est le pessimisme que chaque test introduit ?

Le paragraphe 8.2 présente la méthode de calcul du pire temps de réponse obtenu pour chaque configuration par l'algorithme RM. Le paragraphe 8.3, définit les caractéristiques principales du simulateur. Le paragraphe 8.4, définit les résultats obtenus avec l'analyse de compétitivité et le paragraphe 8.6 expose les résultats de la simulation.

8.2. Calcul exact du pire temps de réponse sous RM

Pour le calcul du pire temps de réponse exact obtenu par l'algorithme RM, nous utilisons le même principe que celui des travaux de [ALZ 05]. En effet, du fait de la présence possible d'anomalies lors de l'ordonnancement d'une configuration sous RM, pour déterminer le pire temps de réponse, nous devons tester pour chaque requête de chaque tâche l'ensemble de toutes les valeurs que peuvent prendre les paramètres.

Lors de la génération d'une tâche, des bornes supérieures du temps d'exécution de chaque sous-tâche et du délai de suspension lui sont attribuées. Les bornes inférieures de ces temps d'exécution et du délai de suspension sont fixées à 1.

Afin de déterminer le pire temps de réponse exact de chaque tâche, un algorithme *brute force* ordonnance la configuration de tâches en respectant la politique d'ordonnancement RM. Il fait varier pour chaque requête de chaque tâche les valeurs des différents paramètres (entre la borne inférieure 1 et celle attribuée à la génération). En considérant toutes ces variations, l'algorithme stocke pour chaque tâche son pire temps de réponse respectant l'échéance de la tâche.

8.3. Caractéristiques des configurations générées

Pour la génération de configurations de tâches à suspension, un simulateur est défini. Mais quelques contraintes doivent être rajoutées au simulateur pour deux raisons principales :

– obtenir un nombre plus important de configurations de tâches à suspension ordonnançables.

– diminuer la complexité du calcul exact du pire temps de réponse obtenu par l'algorithme RM (nous savons que ce problème est $\mathcal{NP}$-Difficile au sens fort, *cf.* chapitre 6).

Ainsi, la première contrainte est de limiter, arbitrairement, la charge processeur des configurations à 0.7. En effet, la diminution de ce paramètre est une donnée importante pour rendre des configurations ordonnançables. Mais une charge processeur inférieure à 0.7 n'assure pas qu'une configuration soit obligatoirement ordonnançable. Une autre contrainte pour obtenir des configurations ordonnançables est de limiter le nombre de tâches de la configuration. Ainsi, ce nombre est limité à trois. C'est sur ce type de configurations que nous obtenons les meilleures bornes du ratio de compétitivité. À noter également que les tâches générées sont

des tâches à échéance sur requête.

De plus, nous avons démontré (*cf.* paragraphe 6.3) la présence d'anomalies pour l'ordonnancement de tâches à suspension avec les algorithmes à priorité fixe. Mais la présence possible d'anomalies d'ordonnancement augmente la complexité de calculs des pires temps de réponse des tâches à suspension. La présence d'anomalies d'ordonnancement impose de ne pas se contenter de calculer le pire temps de réponse des tâches en considérant pour chaque tâche son pire temps de suspension et d'exécution. En effet, diminuer le temps de suspension ou d'exécution d'une tâche peut augmenter le temps de réponse d'une des tâches de la configuration. Par conséquent, pour déterminer le pire temps de réponse d'une tâche d'une configuration, il faut tester toutes les variations possibles des valeurs d'exécution et de suspension pour chacune des requêtes de chacune des tâches à suspension de la configuration.

Afin de limiter le nombre de calculs, il faut limiter le nombre de variations. Ainsi, les configurations de tâches que génère le simulateur ont quelques contraintes supplémentaires :

– limiter les durées d'exécution et de suspension des tâches ;

– limiter la longueur de l'hyperpériode (*cf.* paragraphe 4.1.2.3).

Remarque 10 *C_i (resp. X_i) désigne la limite supérieure du temps processeur requis (resp. la pire durée de suspension) de la tâche τ_i. En conséquence, comme nous nous limitons au temps discret (cf. paragraphe 3), la durée d'exécution (resp. le délai de suspension) d'une tâche prend une valeur entière qui varie entre 1 et la valeur de C_i (resp. X_i). Pour limiter les durées d'exécution et de suspension, les valeurs de $C_{i,1}$, X_i et $C_{i,2}$ sont entières et prennent aléatoirement une valeur comprise dans $\{1, 2, 3, 4\}$.*

Pour réduire la longueur de l'hyperpériode, deux règles sont ajoutées :

– les tâches à suspension générées sont à départ simultané ;

– les tâches sont générées avec des périodes harmoniques (définition 22).

Définition 22 *Soit $I : (\tau_1, \tau_2, \ldots, \tau_n)$ une configuration de tâches. I possède des périodes harmoniques, si et seulement si, les deux propriétés suivantes sont respectées :*

$$T_1 \leq T_2 \leq \cdots \leq T_n$$
$$\forall i, i \in \{2, \ldots, n\}, T_i \mod T_{i-1} = 0$$

Remarque 11 *La première propriété des configurations à période harmonique permet juste de vérifier que les tâches sont ordonnées par ordre croissant sur les périodes. La seconde propriété limite la longueur de l'hyperpériode et ainsi le nombre de requêtes par tâche dans une hyperpériode. En effet, la longueur de l'hyperpériode se limite à la plus grande période (des tâches de la configuration). Or, comme nous étudions l'algorithme à priorité fixe* RM*, la tâche ayant la plus grande période est également la tâche la moins prioritaire.*

Pour la génération d'une configuration de tâches, le simulateur génère les tâches les unes après les autres. Pour chaque tâche τ_i, le simulateur génère :

– dans un premier temps, les valeurs des temps d'exécution : $C_{i,1}$ et $C_{i,2}$ et du délai de suspension X_i ;

– dans un deuxième temps, le simulateur détermine la longueur de la période pour que la charge processeur de la configuration reste inférieure à 0.7. Ainsi, si nous dénotons ϑ, la charge processeur de la configuration avant la génération de la tâche τ_i, pour que la charge reste inférieure à 0.7, il faut :

$$\vartheta + \frac{C_i}{T_i} < 0.7$$

$$T_i > \frac{C_i}{0.7 - \vartheta} \tag{8.1}$$

À noter que la charge est prise strictement inférieure à 0.7, de manière à éviter de diviser par zéro dans la génération de la tâche suivante. Par conséquent, afin de conserver les propriétés de la définition 22, pour calculer la période de la tâche τ_i, nous partons de la période la tâche τ_{i-1}. Puis nous multiplions cette période par un entier aléatoire compris entre 1 et $9.0 - (3 * i) + C_{i,1} + X_i + C_{i,2}$ jusqu'à obtenir une période T_i qui vérifie l'équation 8.1. La formule $9.0 - (3 * i)$ permet de générer une première période relativement importante afin d'éviter une charge processeur dès le départ trop élevée. Puis cette valeur diminue au fur et à mesure de la génération des périodes. La valeur $C_{i,1}+X_i+C_{i,2}$ permet d'obtenir des périodes importantes par rapport aux caractéristiques des tâches. Cette procédure est présentée dans Algorithme 1.

Algorithme 1 : Génération d'une tâche

Donnée : T_{i-1}, la période la tâche précédente ou 1, si c'est la génération de la première tâche.
Donnée : ϑ, la charge processeur de la configuration avant la génération de la tâche τ_i.
Résultat : $\tau_i = (C_{i,1}, X_i, C_{i,2}, T_i)$.
Début
  $C_{i,1} \longleftarrow Random(1, 2, 3, 4)$;
  $X_i \longleftarrow Random(1, 2, 3, 4)$;
  $C_{i,2} \longleftarrow Random(1, 2, 3, 4)$;
  $T_i \longleftarrow T_{i-1}$;
  Tant que $T_i \leq \frac{C_i}{0.7-\vartheta}$ **faire**
    $T_i \longleftarrow T_i * Random(1, .., 9.0 - (3 * i) + C_{i,1} + X_i + C_{i,2})$;
  $\vartheta \longleftarrow \vartheta + \frac{C_i}{T_i}$;
fin

8.4. Bornes inférieures du ratio de compétitivité

Dans ce paragraphe, nous présentons les configurations de tâches à suspension générées menant au pire temps de réponse pour chacun des trois tests d'ordonnançabilité.

Pour déterminer une borne inférieure sur la compétitivité d'un test d'ordonnançabilité pour la minimisation du temps de réponse maximum, bon nombre de configurations de tâches à suspension sont générées (en respectant les caractéristiques imposées dans la paragraphe 8.3).

Pour chaque configuration de tâches, il faut calculer pour chaque tâche :

– la borne supérieure du temps de réponse calculée à l'aide de la formule présentée dans le paragraphe 5.2.

– le temps de réponse maximum exact obtenu par l'algorithme RM.

Pour calculer le temps de réponse exact de chaque tâche d'une configuration, obtenu avec l'algorithme RM, il faut tester toutes les combinaisons possibles des valeurs des temps d'exécution et de suspension des requêtes de chaque tâche.

Puis pour chaque tâche, le ratio de compétitivité est calculé (la borne supérieure, $\sigma(\tau)$, déterminée par le test et divisée par le temps de réponse exact maximum obtenu avec RM, $\sigma_{RM}(\tau)$:

$$\frac{\sigma(\tau)}{\sigma_{RM}(\tau)}$$

Finalement, pour chaque configuration, le plus grand ratio est conservé afin d'obtenir une borne.

8.4.1. *La méthode A de* KIM

Theorème 15 *La limite inférieure du ratio de compétitivité du test d'ordonnançabilité de la méthode A de* KIM *pour la minimisation du temps de réponse maximum lors de l'ordonnancement de tâches à suspension est de* $2,91667$.

Démonstration :

Posons I_A, la configuration à trois tâches à suspension suivante :

$$\tau_1 : C_{1,1} = 3, X_1 = 2, C_{1,2} = 3, T_1 = 12$$
$$\tau_2 : C_{2,1} = 3, X_2 = 1, C_{2,2} = 1, T_2 = 96$$
$$\tau_3 : C_{3,1} = 1, X_3 = 1, C_{3,2} = 1, T_3 = 96$$

La borne supérieure sur le temps de réponse maximum, obtenue pour chaque tâche τ_i de la configuration I_A, par le test d'ordonnançabilité de la méthode A de KIM est égale à :

$$\tau_1 : \sigma_1^A = 8, \tau_2 : \sigma_2^A = 17, \tau_3 : \sigma_3^A = 35$$

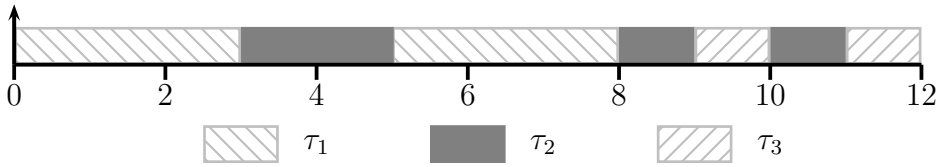

Figure 8.1. *Les temps de réponse exacts obtenus par* RM *en ordonnançant* I_A.

La figure 8.1 présente le temps de réponse maximum exact obtenu avec l'algorithme d'ordonnancement à priorité fixe RM. Pour cette configuration, aucune anomalie d'ordonnancement n'est présente. Ainsi, pour obtenir les pires temps de réponse des tâches, il faut exécuter les tâches de la configuration avec leurs pires durées d'exécution et de suspension.

À l'instant 0, l'algorithme RM ordonnance la tâche τ_1 et durant sa suspension, il commence l'ordonnancement de la tâche τ_2. À la fin de l'ordonnancement de τ_1, à l'instant 8, τ_2 continue son ordonnancement. À l'instant 9, la tâche τ_2 se suspend et pendant sa suspension, la tâche τ_3 est ordonnancée. Finalement, la tâche τ_2 finit son ordonnancement à l'instant 11, et la tâche τ_3, à l'instant 12. Par conséquent, les temps de réponse maximums des tâches (dénotés σ_i^{RM}) valent :

$$\tau_1 : \sigma_1^{RM} = 8, \tau_2 : \sigma_2^{RM} = 11, \tau_3 : \sigma_3^{RM} = 12$$

Par conséquent, le ratio de compétitivité obtenu par la méthode A de KIM sur cette configuration I_A est de :

$$\begin{aligned} c_A^{RM} &= \sup_{anyI} \frac{\sigma_A(I)}{\sigma_{RM}(I)} \geq \frac{\sigma_A(I_A)}{\sigma_{RM}(I_A)} \\ &\geq \max\left(\frac{\sigma_1^A}{\sigma_1^{RM}}, \frac{\sigma_2^A}{\sigma_2^{RM}}, \frac{\sigma_3^A}{\sigma_3^{RM}}\right) \\ &\geq \frac{\sigma_3^A}{\sigma_3^{RM}} = \frac{35}{12} = 2.91667 \end{aligned}$$

Ainsi, la borne inférieure du ratio de compétitivité de la méthode A de KIM est fixée pour le moment à 2.91667.

□

8.4.2. *La méthode B de Kim*

Theorème 16 *La borne inférieure du ratio de compétitivité de la méthode B de Kim pour la minimisation du temps de réponse maximum est de* $2,75$.

Démonstration :

Soit I_B, la configuration de tâches à suspension suivante :

$$\tau_1 : C_{1,1} = 1, X_1 = 1, C_{1,2} = 3, T_1 = 6$$
$$\tau_2 : C_{2,1} = 1, X_2 = 3, C_{2,2} = 2, T_2 = 270$$
$$\tau_3 : C_{3,1} = 3, X_3 = 2, C_{3,2} = 3, T_3 = 810$$

La borne supérieure du temps de réponse maximum calculée par la méthode B de KIM pour chaque tâche τ_i de la configuration de tâches à suspension I_B et dénotée σ_i^B est égale à :

$$\tau_1 : \sigma_1^B = 5, \tau_2 : \sigma_2^B = 22, \tau_3 : \sigma_3^B = 35$$

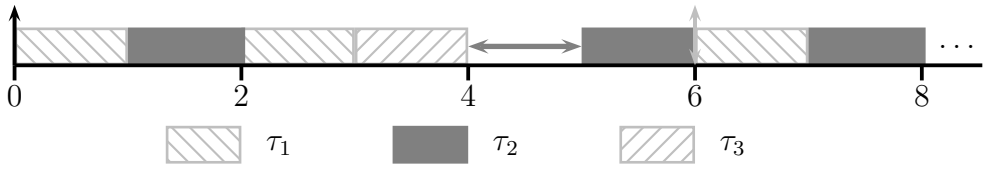

Figure 8.2. *Ordonnancement de la configuration I_B par l'algorithme* RM.

Pour la configuration I_B, le pire ratio de compétitivité est obtenu avec la tâche τ_2. Ainsi, la figure 8.2 présente l'ordonnancement de la configuration I_B permettant d'obtenir le temps de réponse maximum exact de la tâche τ_2. Pour obtenir ce temps de réponse maximum, la tâche τ_1 doit être exécutée avec un temps d'exécution $C_{1,2} = 1$. Les autres paramètres de la configuration sont à leur valeur maximum. À l'instant 0, l'algorithme RM ordonnance τ_1 car elle est la tâche avec la priorité la plus élevée. À l'instant 1, la tâche τ_1 se suspend et la tâche τ_2 est ordonnancée. À l'instant 2, τ_2 est suspendue et la tâche τ_1 finit son ordonnancement (à l'instant 3). À l'instant 3, la tâche τ_3 est ordonnancée pendant la suspension de la tâche τ_2. À l'instant 6, la seconde requête de la tâche τ_1 se réveille et est exécutée. À l'instant 7, la tâche τ_2 est ordonnancée et finit son exécution à l'instant 8. Par conséquent, nous obtenons les temps de réponse exacts des tâches suivants :

$$\tau_1 : \sigma_1^{RM} = 5, \tau_2 : \sigma_2^{RM} = 8, \tau_3 : \sigma_3^{RM} = 24$$

Par conséquent, le ratio de compétitivité de la méthode B de KIM est de :

$$\begin{aligned} c_B^{RM} &= \sup_{anyI} \frac{\sigma_B(I)}{\sigma_{RM}(I)} \geq \frac{\sigma_B(I_B)}{\sigma_{RM}(I_B)} \\ &\geq \max\left(\frac{\sigma_1^B}{\sigma_1^{RM}}, \frac{\sigma_2^B}{\sigma_2^{RM}}, \frac{\sigma_3^B}{\sigma_3^{RM}}\right) \\ &\geq \frac{\sigma_2^B}{\sigma_2^{RM}} = \frac{22}{8} = 2.75 \end{aligned}$$

□

8.4.3. *La méthode de Jane W. S.* LIU

Theorème 17 *Le ratio de compétitivité obtenu par la méthode de* LIU *est de* $2,875$ *pour la minimisation du temps de réponse maximum lorsque les tâches se suspendent au plus une fois*

Démonstration :

Pour cette démonstration, nous réutilisons la configuration I_B définie dans la démonstration de la borne inférieure de la méthode B de Kim (Théorème 16).

La borne supérieure du temps de réponse maximum obtenue avec la méthode de LIU pour chaque tâche τ_i de la configuration I_B et dénotée σ_i^L est de :

$$\tau_1 : \sigma_1^L = 5, \tau_2 : \sigma_2^L = 23, \tau_3 : \sigma_3^L = 47$$

La figure 8.2 présente le temps de réponse maximum exact obtenu pour la tâche τ_2, avec l'algorithme RM en ordonnançant I_B. Ces temps de réponse sont les mêmes que ceux obtenus dans la démonstration de la borne inférieure de la méthode B de KIM :

$$\tau_1 : \sigma_1^{RM} = 5, \tau_2 : \sigma_2^{RM} = 8, \tau_3 : \sigma_3^{RM} = 24$$

Par conséquent, le ratio de compétitivité obtenu avec la méthode de LIU pour la minimisation du temps de réponse maximum est égal à :

$$\begin{aligned} c_L^{RM} &= \sup_{anyI} \frac{\sigma_L(I)}{\sigma_{RM}(I)} \geq \frac{\sigma_L(I_B)}{\sigma_{RM}(I_B)} \\ &\geq \max\left(\frac{\sigma_1^L}{\sigma_1^{RM}}, \frac{\sigma_2^L}{\sigma_2^{RM}}, \frac{\sigma_3^L}{\sigma_3^{RM}}\right) \\ &\geq \frac{\sigma_2^L}{\sigma_2^{RM}} = \frac{23}{8} = 2.875 \end{aligned}$$

□

Nous avons également déterminé un dernier test d'ordonnançabilité : la "*meilleure méthode*".

8.4.4. *La meilleure méthode*

Cette méthode consiste à appliquer *pour chaque tâche* d'une configuration les trois tests d'ordonnançabilité précédemment étudiés. Pour chaque tâche, la valeur retenue pour la borne supérieure du temps de réponse maximum est la plus petite des valeurs obtenues par les tests d'ordonnançabilité. Une telle méthode peut permettre de diminuer la valeur du ratio de compétitivité mais nous ne disposons pas de démonstration formelle à cette affirmation. Mais, le ratio de compétitivité de cette "meilleure méthode" est inférieur à celui des autres tests d'ordonnançabilité :

Theorème 18 *Le ratio de compétitivité de la meilleure méthode pour la minimisation du temps de réponse maximum pendant l'ordonnancement de tâches à suspension est de* $2,16667$.

Démonstration :

Posons I_C, la configuration de tâches à suspension suivante :

$$\tau_1 : C_{1,1} = 1, X_1 = 1, C_{1,2} = 3, T_1 = 9$$
$$\tau_2 : C_{2,1} = 1, X_2 = 3, C_{2,2} = 1, T_2 = 72$$
$$\tau_3 : C_{3,1} = 3, X_3 = 2, C_{3,2} = 1, T_3 = 648$$

Le temps de réponse maximum exact obtenu pour chaque tâche avec l'algorithme d'ordonnancement RM est de :

$$\tau_1 : \sigma_1^{RM} = 5, \tau_2 : \sigma_2^{RM} = 6, \tau_3 : \sigma_3^{RM} = 14$$

Le tableau 8.1 présente pour chacune des trois tâches de la configuration I_C, le temps de réponse maximum obtenu pour chaque test d'ordonnançabilité.

Tâches	Méthode A de KIM	Méthode B de KIM	Méthode de LIU
τ_1	5	5	5
τ_2	13	13	14
τ_3	22	20	23

Tableau 8.1. *Temps de réponse maximums calculés pour chaque tâche de I_C et chaque test*

Pour chaque tâche, le tableau 8.2 présente les temps de réponse maximums obtenus avec la meilleure méthode :

Tâches	Meilleure méthode
τ_1	5
τ_2	13
τ_3	20

Tableau 8.2. *Meilleur temps de réponse maximum pour chaque tâche de I_C*

Par conséquent, le ratio de compétitivité de la meilleure méthode, C_{Bst}, est de :

$$\begin{aligned} c_{Bst}^{RM} &= \sup_{anyI} \frac{\sigma_{Bst}(I)}{\sigma_{RM}(I)} \geq \frac{\sigma_L(I_C)}{\sigma_{RM}(I_C)} \\ &\geq \sup_{1 \leq i \leq 3} \{\inf\{c_A^{RM}(\tau_i), c_B^{RM}(\tau_i), c_L^{RM}(\tau_i)\}\} \\ &\geq 2.16667 \end{aligned}$$

□

8.5. Incomparabilité des tests de faisabilité

Ces résultats tendent à montrer que la méthode B de KIM est celle qui obtient les meilleurs résultats. Mais, en réalité, nous ne pouvons pas conclure que la méthode B soit la meilleure. En effet, la seule conclusion possible est que ces tests d'ordonnançabilité ne sont pas comparables. Aucune conclusion n'est possible car il est possible pour chaque test d'ordonnançabilité de définir une configuration de tâches où le test concerné est le meilleur des trois tests sur cette configuration. KIM *et al.* ont déjà prouvé que leurs deux méthodes ne sont pas comparables [KIM 95]. Maintenant, nous allons démontrer que la méthode A de *Kim* et la méthode de *Liu* ne sont pas comparables : en exhibant deux configurations de tâches, où les méthodes comparées sont tour à tour la meilleure.

Soit I la configuration de tâches suivante :

$$\tau_1 : C_{1,1} = 2, X_1 = 3, C_{1,2} = 1, T_1 = 7$$
$$\tau_2 : C_{2,1} = 1, X_2 = 3, C_{2,2} = 2, T_2 = 56$$
$$\tau_3 : C_{3,1} = 3, X_3 = 1, C_{3,2} = 2, T_3 = 392$$

Le ratio de compétitivité obtenu pour ces deux tests d'ordonnançabilité sont :

$$\sigma_A^{RM}(I) = 1.47$$
$$\sigma_L^{RM}(I) = 1.80$$

Le ratio obtenu avec la méthode A de *Kim* est meilleur que le ratio obtenu par la méthode de LIU.

Soit I' la configuration de tâches à suspension suivante :

$$\tau_1 : C_{1,1} = 2, X_1 = 3, C_{1,2} = 1, T_1 = 9$$
$$\tau_2 : C_{2,1} = 2, X_2 = 3, C_{2,2} = 3, T_2 = 45$$
$$\tau_3 : C_{3,1} = 2, X_3 = 2, C_{3,2} = 1, T_3 = 90$$

Les ratios obtenus sont maintenant de :

$$\sigma_A^{RM}(I') = 1.69$$
$$\sigma_L^{RM}(I') = 1.56$$

Mais sur cette configuration, la méthode de LIU obtient un meilleur ratio que celui obtenu par la méthode A de KIM. Ainsi, ces tests ne sont pas comparables entre eux.

Nous ne pouvons pas nous prononcer quant à savoir si la méthode B de KIM et la méthode de LIU sont comparables car avec notre générateur de tâches, nous avons déterminé des configurations où la méthode B de KIM est meilleure que celle de LIU. Cependant, nous n'avons jamais généré une configuration telle que la méthode de LIU soit meilleure que la méthode B de KIM. Par conséquent, il se peut qu'il existe une telle configuration mais nous n'en savons en réalité rien.

8.6. Étude par la simulation

8.6.1. *Introduction*

Dans ce paragraphe, nous présentons les résultats numériques obtenus en générant avec l'algorithme brute force (décrit dans le paragraphe 8.3), un bon nombre de configurations de tâches à suspension. Tous les tests d'ordonnançabilité (les deux méthodes de KIM, la méthode de LIU) et l'algorithme de calcul du temps de réponse maximum exact de l'algorithme RM, ont été appliqués à toutes les configurations de tâches générées. Ensuite pour chaque configuration, le ratio de compétitivité pour la minimisation du temps de réponse maximum est calculé. Finalement, le ratio pour chaque test et chaque configuration est stocké puis des statistiques sur ces ratios sont calculés pour chaque test d'ordonnançabilité.

Nous savons que l'environnement de notre simulation (en particulier l'affectation des caractéristiques des tâches) n'est pas suffisant (*cf.* [BIN 04]) pour déterminer des comparaisons définitives et exhaustives sur les différents tests d'ordonnançabilité. Par conséquent, nos conclusions ne sont valables que dans le confinement de notre modèle stochastique (*cf.* paragraphe 8.3). Nous vous rappelons que le simulateur a été développé pour définir des bornes inférieures de ratio de compétitivité et non pour mener une étude des tests par simulation.

8.6.2. *Résultats*

Pour obtenir des résultats pertinents d'un point de vue statistique, un million de configurations de tâches à suspension ont été générées. Les configurations de tâches à suspension générées respectent également les contraintes imposées dans le paragraphe 8.3. Le tableau 8.3 représente les résultats statistiques obtenus par chacun des trois tests d'ordonnançabilité étudiés.

Trois caractéristiques statistiques ont été calculées après la génération du million de configurations et l'application de chaque test sur chaque configuration :

– La première ligne du tableau 8.3 présente pour chaque test le pourcentage de fois où chaque test d'ordonnançabilité a été le meilleur des trois tests d'ordonnançabilité pour une configuration (c.-à-d. qu'il a obtenu le plus petit ratio de compétitivité). La méthode B de KIM est celle obtenant le meilleur pourcentage loin devant la méthode A de KIM.

– La moyenne du ratio de compétitivité obtenue pour chaque test est stockée dans la seconde ligne du tableau 8.3. La méthode B de KIM est celle obtenant le ratio moyen le plus petit. À noter également que même si la méthode de LIU n'est quasiment pour chaque configuration jamais la meilleure (*cf.* la première ligne du tableau), elle n'en dispose pas moins d'un ratio moyen meilleur à celui de la méthode A de KIM.

– Avec l'écart type (troisième ligne), le meilleur des trois tests d'ordonnançabilité est celui de la méthode A de KIM même si les résultats sont globalement proches.

tests d'ordonnançabilité	Méthode A de KIM	Méthode B de KIM	Méthode de LIU
Pourcentage	3.64%	99.8%	$\approx$ 0.00%
Moyenne	1.65	1.21	1.50
Écart type	0.18	0.20	0.22

Tableau 8.3. *Résultats de la simulation pour les tests d'ordonnançabilité pour des configurations de tâches composées de* 2 *ou* 3 *tâches*

8.7. Conclusion

Dans ce chapitre, les tests d'ordonnançabilité (calculant une borne supérieure du temps de réponse maximum) basés sur l'algorithme à priorité fixe RM, ont été étudiés. Pour chacun de ces trois tests, une borne inférieure du ratio de compétitivité (pour la minimisation du temps de réponse maximum) a été établie. Ces ratios de compétitivité permettent de représenter le pessimisme qu'engendre chacun de ces tests par rapport à l'algorithme d'ordonnancement RM. La borne inférieure la plus petite étant égale à 2.75, c'est la méthode B de KIM. La borne inférieure de la méthode A de KIM est de 2.91667 et celle de la méthode de LIU : 2.875. Mais il faut rappeler également que dans le chapitre 7, nous avons démontré que l'algorithme d'ordonnancement RM n'est pas non plus un algorithme optimal et que son ratio de compétitivité par rapport à un adversaire clairvoyant est de au moins 2, ceci montre que valider un système temps réel en présence de tâches à suspension avec ces méthodes peut conduire à surdimensionner des systèmes de façon importante. Ainsi, ces tests d'ordonnançabilité introduisent une part de pessimisme de par leurs calculs mais aussi parce qu'ils sont basés sur l'algorithme RM qui n'est que 2-compétitif.

À noter également qu'en utilisant l'analyse de compétitivité ainsi que la simulation, le meilleur des tests d'ordonnançabilité est celui de la méthode B de KIM, même si ces résultats sont uniquement valables dans le cadre de notre modèle stochastique utilisé pour la génération de configurations de tâches. Finalement, les résultats obtenus avec ces trois tests d'ordonnançabilité ont été améliorés en considérant pour chaque tâche de chaque configuration une combinaison des tests présentés (*cf.* la meilleure méthode, paragraphe 8.4.4). Le ratio de compétitivité a alors atteint la valeur de 2.166667.

Ces résultats ont été publiés dans une conférence, [RID 06].

8.8. Bibliographie

[ALZ 05] ALZEER I., MOLINARO P., TRINQUET Y., « Calcul exhaustif du Temps de Réponse de tâches et messages dans un système temps réel réparti », *In Proceedings of the* $13^{t}h$ *Real-Time Systems*, 2005.

[BIN 04] BINI E., BUTTAZZO G., « Biasing Effects in Schedulability Measures », *Proceedings of the 16th Euromicro Conference on Real-Time Systems, Catania, Italy*, page , June 2004.

[CHE 96] CHEN B., VESTJENS A., « Scheduling on identical machines : How good is LPT in an on-line setting ? », *Operations Research Letters*, , n°13, March 1996.

[EPS 01] EPSTEIN L., VAN-STEE R., « Lower bounds for on-line single machine scheduling », *proc. 26th Mathematical Foundations of Computer Science*, p. 338-350, 2001.

[HOO 96] HOOGEVEEN H., VESTJENS A., « Optimal on-line algorithms for single-machine scheduling », *in : proc. 5th Conference on Integer Programming and Combinatorial Optimization*, p. 404-414, 1996.

[KIM 95] KIM I., CHOI K., PARK S., KIM D., HONG M., « Real-Time Scheduling of tasks that contain the external blocking intervals », *proc. Conference on Real-Time Computing Systems and Applications*, p. 54-59, 1995.

[LIU 00] LIU J., *Real-Time Systems*, Prentice hall, 2000.

[RID 06] RIDOUARD F., RICHARD P., « Worst-case analysis of feasibility tests for self-suspending tasks », *proc. International Conference on Real-Time and Network Systems (RTNS 2006), Poitiers*, vol. 1, n°30-31, p. 15–24, mai

Chapitre 9

Conclusion sur les tâches à suspension

Nous avons étudié dans cette partie le problème d'ordonnancement des tâches à suspension dans un contexte monoprocesseur. Nous avons axé notre étude sur les tâches qui ne peuvent se suspendre qu'au plus une seule fois. Cette recherche s'est portée sur trois domaines :

– la difficulté de l'ordonnancement des tâches à suspension sur un système monoprocesseur ;

– la compétitivité des algorithmes en-ligne classiques selon deux critères de performance : la minimisation du nombre de tâches en retard et la minimisation du temps de réponse maximum ;

– l'évaluation du pessimisme engendré par les tests d'ordonnançabilité conçus pour l'ordonnancement des tâches à suspension.

Il avait déjà été prouvé (*cf.* [RIC 03]) que l'ordonnancement de tâches à suspension périodiques et à départ simultané est un problème $\mathcal{NP}$-Difficile au sens fort. Nous avons démontré dans cette étude que le problème ouvert de complexité où les tâches sont à échéance sur requête est également un problème $\mathcal{NP}$-Difficile au sens fort. De plus, nous avons conclu qu'il n'existe pas d'algorithme *universel* pour ordonnancer des tâches avec suspension, sauf si $\mathcal{P} = \mathcal{NP}$.

Une autre difficulté d'ordonnancement démontrée est la présence possible d'anomalies lors de l'ordonnancement de tâches à suspension par les algorithmes classiques d'ordonnancement RM, DM, EDF (*cf.* paragraphe 4.2).

Nous avons également établi qu'aucun algorithme en-ligne déterministe n'est optimal pour l'ordonnancement de tâches à suspension dans un système monoprocesseur.

En utilisant l'analyse de compétitivité, nous avons évalué les performances des algorithmes en-ligne classiques (RM, DM, EDF et LLF (paragraphe 4.2) pour l'étude de deux critères de performance : la minimisation du nombre de tâches en retard et la minimisation du temps de réponse maximum. Ainsi, nous avons démontré que les algorithmes classiques en-ligne sont non compétitifs pour minimiser le nombre de tâches en retard. En particulier, ils sont

même non compétitifs sur des configurations de tâches à suspension avec des charges processeurs presque nulles alors qu'il existe des ordonnancements faisables de ces configurations générées. Pour ce même problème, nous avons également démontré que la technique d'augmentation des ressources n'amène aucune amélioration pour l'ordonnancement de tâches à suspension sous EDF. Quant à la minimisation du temps de réponse maximum, nous avons établi que les algorithmes en-ligne classiques ne sont pas mieux que 2-compétitifs.

Nous avons finalement étudié trois tests d'ordonnançabilité pour les tâches à suspension, basés sur l'algorithme à priorité fixe RM. Ces trois tests sont les méthodes A et B de KIM [KIM 95] et la méthode de LIU [LIU 00]. Cette étude a eu pour but d'évaluer le pessimisme qu'ils induisent. Pour ces trois tests, en utilisant l'analyse de compétitivité, une borne inférieure du ratio de compétitivité pour la minimisation du temps de réponse maximum a été établie. Elle est de 2.91667 pour la méthode A de KIM, 2.75 pour la méthode B et 2.875 pour la méthode de LIU. Ces résultats ont été améliorés en utilisant la *meilleure méthode* qui consiste à considérer pour chaque tâche de chaque configuration, celui des trois tests qui mène au plus petit ratio. Le ratio de cette méthode a été établi à 2.166667. Nous avons également prouvé qu'il n'y avait aucune équivalence entre la méthode A de KIM et la méthode de LIU. En utilisant la simulation, les résultats de l'analyse de compétitivité ont été confirmés et la méthode B de KIM est celle qui obtient les meilleurs résultats. Toutefois, l'ensemble de ces résultats sur ces tests d'ordonnançabilité sont valables dans le cadre de notre modèle stochastique utilisé pour la génération des configuration de tâches.

Ces résultats ont fait l'objet de plusieurs publications. La conférence [RID 04] a présenté la complexité du problème ainsi que la présence des anomalies d'ordonnancement sous la politique EDF et la compétitivité des algorithmes en-ligne pour la minimisation du nombre de tâches en retard. Dans [RID 05], nous avons présenté en plus, les résultats sur la compétitivité pour la minimisation du temps de réponse maximum. L'ensemble de ces résultats a été regroupé dans la revue internationale [RID 06b]. Finalement la conférence [RID 06a] présente le résultat de non optimalité des algorithmes en-ligne et l'étude sur les tests d'ordonnançabilité.

Pour de futurs travaux, il serait intéressant de rechercher un algorithme hors-ligne optimal pour l'ordonnancement de tâches à suspension. Il faudrait aussi mesurer l'impact des anomalies d'ordonnancement sur le nombre de tâches générant le non respect de contraintes temporelles, en utilisant le principe présenté sur les tâches non préemptibles et sans suspension [MOK 05]. Il serait également intéressant d'étudier les deux sous problèmes déduis du problème des tâches à suspension : le problème de gigue sur activation et le problème de délai de livraison. Une dernière piste à envisager est d'étendre le problème d'ordonnancement des tâches à suspension aux tâches dépendantes ainsi qu'aux systèmes multiprocesseurs.

9.1. Bibliographie

[KIM 95] KIM I., CHOI K., PARK S., KIM D., HONG M., « Real-Time Scheduling of tasks that contain the external blocking intervals », *proc. Conference on Real-Time Computing Systems and Applications*, p. 54-59, 1995.

[LIU 00] LIU J., *Real-Time Systems*, Prentice hall, 2000.

[MOK 05] MOK A., POON W.-C., « Non-Preemptive Robustness under Reduced System Load », 26^{th} *IEEE International Real-Time Systems Symposium (RTSS'05)*, vol. 1, n°5-8, p. 200-209, December 2005.

[RIC 03] RICHARD P., « On the complexity of scheduling real-time tasks with self-suspensions on one processor », *IEEE Euromicro Conf. on Real-Time Systems (ECRTS'03)*, page 8p, 2003.

[RID 04] RIDOUARD F., RICHARD P., COTTET F., « Negative results for scheduling independent hard real-time tasks with self-suspensions », 25^{th} *IEEE International Real-Time Systems Symposium (RTSS'04)*, 2004.

[RID 05] RIDOUARD F., RICHARD P., COTTET F., « Ordonnancement des tâches indépendantes avec suspension », *proc. Real-Time Embedded Systems (RTS'05), Paris*, 2005.

[RID 06a] RIDOUARD F., RICHARD P., « Worst-case analysis of feasibility tests for self-suspending tasks », *proc. International Conference on Real-Time and Network Systems (RTNS 2006), Poitiers*, vol. 1, n°30-31, p. 15–24, mai

[RID 06b] RIDOUARD F., RICHARD P., COTTET F., TRAORE K., « Some results on scheduling tasks with self-suspensions », *Journal of Embedded Computing (JEC), to appear*, 2006.

Deuxième partie :

Ordonnancement par une machine à traitement par lot

DEUXIÈME PARTIE

Ordonnancement par une machine à traitement par lot

Cette partie décrit nos résultats sur l'ordonnancement classique par une machine à traitement par lot :

– Le chapitre 10 introduit les machines à traitement par lot et décrit le problème étudié.

– Le chapitre 11 expose un état de l'art de ce problème d'ordonnancement. Ce chapitre présente les résultats des problèmes hors-ligne et en-ligne.

– Le chapitre 12 décrit nos résultats et détaille l'ensemble des algorithmes que nous proposons.

– En conclusion, au chapitre 13, nous rappelons nos résultats.

Table des matières

Chapitre 10

Introduction sur les machines à traitement par lot

10.1. Contexte

Le travail effectué dans cette partie s'inscrit dans le projet industriel W4L (*Workload For Labs*). Les laboratoires pharmaceutiques analysent des échantillons pour leurs clients. Pour le bon fonctionnement du laboratoire, les activités d'analyses sont planifiées. La gestion des analyses est informatisée, mais aucun logiciel ne propose une automatisation de la planification des activités qui permettrait d'améliorer la réactivité ainsi que les performances du laboratoire. Le projet W4L avait pour objectif de combler ce vide. Chaque demande d'analyse d'un client se retranscrit dans le laboratoire par un ensemble d'échantillons. Nous considérons que chaque échantillon suit les mêmes analyses. Chaque demande d'un client se traduit pour notre modélisation par une tâche et chaque analyse par une opération. Les opérations associées à un même ensemble d'analyses peuvent être réalisées simultanément, les machines traitant parallèlement plusieurs analyses (p. ex. une centrifugeuse). Ces machines sont appelées *machines à traitement par lot*. Le but de notre travail qui s'inscrit dans un problème théorique en dehors du contrat lui-même, est d'étudier l'ordonnancement des échantillons sur une telle machine.

Les machines à traitement par lot permettent d'ordonnancer plusieurs tâches simultanément. Les tâches ordonnancées ainsi simultanément, forment des *lots*. D'autres domaines utilisent aussi ces machines comme, par exemple, les usines de fabrication de circuits intégrés (détaillé dans [LEE 99]).

L'objectif des machines à traitement par lot est de regrouper l'exécution des tâches qui arrivent dans le système dans un unique lot d'exécution. Le choix de lancer l'exécution d'un lot est pris par l'ordonnanceur si le nombre de tâches actives et en attente est supérieur ou égal à la capacité des lots. Une autre possibilité est que l'ordonnanceur se donne un temps pour récupérer toutes les tâches actives et les exécuter à la fin de ce laps de temps.

Suivant les modalités d'ordonnancement des lots, il existe deux types de machines à traitement par lot :

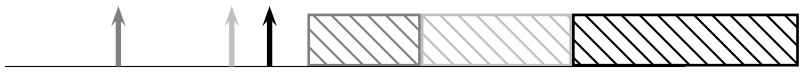

Figure 10.1. *Ordonnancement d'une machine à traitement par lot en série*

– *Les machines à traitement en série* : les tâches sont exécutées les unes après les autres. Ainsi le temps processeur du lot est égal à la somme des temps processeurs des tâches composant le lot. Cet ordonnancement est présenté par la figure 10.1. Chaque tâche, quand elle arrive, est mise en attente, et lorsque le système estime avoir assez attendu, il lance l'exécution des tâches qui s'exécutent les unes après les autres.

– *Les machines à traitement parallèle* : les tâches sont exécutées parallèlement. Ainsi le temps processeur du lot est égal au maximum des temps processeurs composant le lot. La figure 10.2 présente cet ordonnancement.

10.2. Le problème étudié : Ordonnancement par une machine à traitement par lot

Notre travail s'est concentré uniquement sur l'analyse des machines à traitement parallèle. Par abus de langage, nous appellerons de telles machines des machines à traitement par lot, sans préciser le parallélisme. Maintenant, nous définissons quelques propriétés sur les machines à traitement par lot :

– La capacité d'un lot se note b, elle peut être finie ou infinie (*cf.* figure 10.2).

– Aucune tâche ne peut préempter un lot en exécution.

– Aucune tâche ne peut quitter un lot au cours de son exécution.

– La longueur d'exécution d'un lot est égale au plus long temps d'exécution des tâches (comme le montre la figure).

Remarque 12 *A noter que la dernière propriété n'est qu'une conséquence des précédentes.*

Remarque 13 *La capacité b des lots de la machine, si elle vaut 1 revient à faire des machines à traitement par lot de l'ordonnancement monoprocesseur. Ce qui n'aurait aucun intérêt dans notre étude. Par conséquent, la capacité des machines à traitement par lot sera d'au moins 2.*

Nous concentrons notre étude sur l'ordonnancement par des algorithmes en-ligne de tâches par les machines à traitement par lot. De plus, le critère d'optimisation étudié, est celui de la minimisation de la longueur de l'ordonnancement. Le système considéré est un système monoprocesseur. Les tâches ne sont pas supposées à départ simultané et la capacité de la machine est considérée comme infinie, si le cas contraire n'est pas précisé. Ainsi à l'aide de

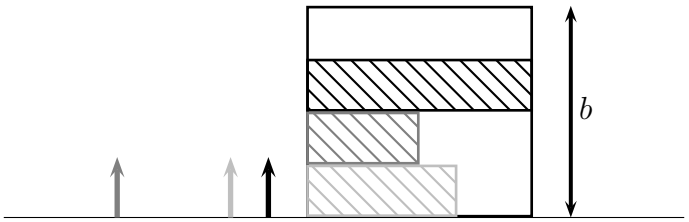

Figure 10.2. *Ordonnancement d'une machine à traitement par lot parallèle*

la notation à trois champs de GRAHAM et al. [GRA 96] (*cf.* Paragraphe 2.3.3), le problème d'ordonnancement étudié se note :

$$1 \mid p-batch, r_i, b = \infty \mid C_{max} \tag{10.1}$$

10.3. Notations sur les lots

Pour l'étude des machines à traitement par lot, précisons quelques notations :

Pour l'ensemble des tâches considérées, notre modèle de tâches se limite à celui à deux paramètres : la date d'arrivée et le temps processeur. Ainsi, en particulier, les tâches ne disposent pas d'échéance. De plus, les tâches sont rangées par ordre chronologique d'activation. C'est-à-dire que si nous notons une configuration I comportant n tâches : $I = \{\tau_1, \ldots, \tau_n\}$. Alors, pour tout i et j tels que $1 \leq i < j \leq n$ la formule suivante est vérifiée :

$$i \leq j \Rightarrow r_i \leq r_j$$

La séquence d'exécution de l'ordonnancement par un algorithme en-ligne d'une configuration de tâches se définira par la création puis l'exécution de m lots (avec $m > 0$). Les tâches sont ordonnancées en lots et m définit le nombre de lots créés et exécutés pour l'ordonnancement d'une configuration.

Pour chaque configuration, les tâches sont ordonnancées dans des lots. Dans la séquence d'exécution résultant de l'ordonnancement par un algorithme en-ligne d'une configuration de tâches, nous noterons par m le nombre de lots générés puis ordonnancés, comme chaque configuration est composée d'au moins une tâche, $m > 0$. Les lots ainsi générés sont notés par ordre de génération chronologique : B_1,...,B_m. Chaque lot B_i ($1 \leq i \leq m$) démarre son exécution à un instant noté s_i. De plus, dans chacun des lots générés, B_i ($1 \leq i \leq m$), la tâche ayant le plus grand temps d'exécution requis et qui par conséquent détermine la longueur d'exécution du lot est notée : $\tau_{(i)}$ (sa date d'arrivée et son temps processeur sont ainsi dénommés : $r_{(i)}$ et $C_{(i)}$).

Le critère de minimisation est celui de la longueur de l'ordonnancement. Avec les notations précédentes, il revient à minimiser la date de fin d'exécution du dernier lot : B_m. Ce qui revient à minimiser la valeur : $s_m + C_{(m)}$.

	Propositions			Bibliographie	
Algorithmes	αH	$\alpha H2$	αH^{∞}	H^{∞}	DENG
Temps processeurs égaux	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$
Temps processeurs agréables	$\frac{1+\sqrt{5}}{2}$	2	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$
Deux dates d'activation	$\frac{1+\sqrt{5}}{2}$	2	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$
Cas général	2	2	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$	$\frac{1+\sqrt{5}}{2}$

Tableau 10.1. *Présentation des différents résultats présents dans cette partie sur les machines à traitement par lot de capacité infinie.*

10.4. Plan

Le paragraphe 11 décrit un état de l'art de notre problème d'ordonnancement : la minimisation de la longueur d'exécution pour l'ordonnancement par une machine à traitement par lots. Notamment, nous proposons deux algorithmes : l'algorithme de DENG et H^{∞} qui font partie des meilleurs algorithmes en-ligne (*cf.* tableau 10.1). Le paragraphe 12 s'étend sur la contribution de nos travaux dans ce domaine. Deux algorithmes en-ligne, αH et $\alpha H2$ ont été établis, ils ne font pas partie des meilleurs algorithmes en-ligne pour le problème général mais ont des résultats intéressants dans certains cas particuliers. En effet, comme le représente le tableau 10.1, ces algorithmes ont des résultats positifs si les temps processeurs sont égaux ou agréables ou si encore il n'y a que deux dates d'activation distinctes. L'algorithme αH^{∞}, notre dernier algorithme, fait partie des meilleurs algorithmes en-ligne pour le problème général. Cet algorithme obtient toujours un délai d'attente plus petit par rapport aux algorithmes représentés dans le tableau 10.1. À la fin de ce chapitre, nous présentons dans le paragraphe 12.7, un dernier algorithme qui donne un panel de solutions au problème d'ordonnancement en-ligne : $1|p - batch, r_i, b = \infty|C_{max}$.

10.5. Bibliographie

[GRA 96] GRAHAM R., LAWLER E., LENSTRA J., KAN A. R., « Optimisation and approximation in deterministic sequencing and scheduling : A survey », *Annals of Discrete Mathematics*, vol. 5, p. 287-326, 1996.

[LEE 99] LEE C., UZSOY R., « Minimizing makespan on a single batch processing machine with dynamic job arrivals », *International Journal of Production Research*, vol. 37, n°1, p. 219-236, 1999.

Chapitre 11

Ordonnancement de machines à traitement par lot : État de l'art

11.1. Introduction

Après la présentation de quelques propriétés sur les lots, nous faisons apparaître l'ensemble des résultats établis concernant le problème hors-ligne et le problème en-ligne quant à la minimisation de la longueur de la séquence pour les machines à traitement par lot.

11.2. Propriétés sur les lots

Définition 23 *Un lot B_i est dit régulier si et seulement si le début de l'exécution de ce lot n'a pas été retardé par la fin des exécutions des lots qui le précèdent. Ce type de lot est l'opposé des lots dit "retardés" (cf. figure 11.1).*

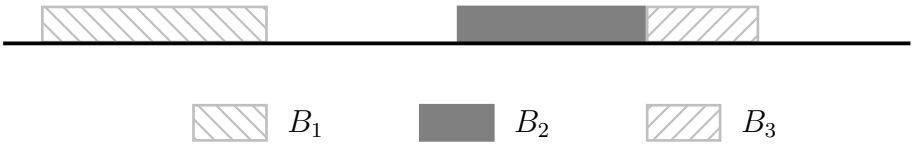

Figure 11.1. *Ordonnancement des lots : a. Le lot B_1 est régulier ; b. Le lot B_3 est retardé*

Remarque 14 *Pour n'importe quel algorithme, le premier lot ordonnancé ne peut pas être retardé par l'exécution d'un autre lot, il est donc toujours régulier.*

Remarque 15 *Ainsi, pour n'importe quel algorithme, il existe toujours dans sa séquence d'exécution, un dernier lot régulier tel que tous les autres lots qui le suivent sont ordonnancés sans insérer de temps creux dans la séquence (c.-à-d. sans attente).*

Définition 24 *Un bloc définit un ensemble de lots qui s'exécutent les uns après les autres sans insérer le temps creux. Ainsi, le premier lot est régulier et les autres lots du bloc sont retardés.*

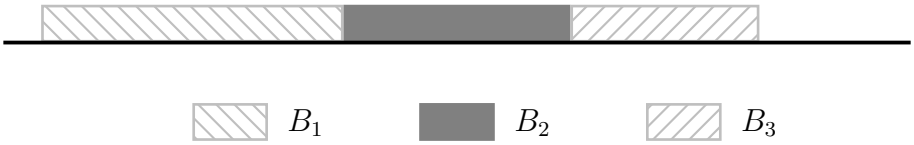

Figure 11.2. *Un bloc de lots*

Pour chaque configuration, les tâches sont ordonnancées dans des lots. Les lots ainsi générés sont notés par ordre de génération chronologique : B_1,...,B_m. Chaque lot B_i ($1 \leq i \leq m$) démarre son exécution à un instant noté s_i. De plus, dans chacun des lots générés, B_i ($1 \leq i \leq m$), la tâche ayant le plus grand temps d'exécution requis et qui par conséquent détermine la longueur d'exécution du lot est notée : $\tau_{(i)}$ (sa date d'arrivée et son temps processeur sont ainsi dénommés : $r_{(i)}$ et $C_{(i)}$).

Le critère de minimisation est celui de la longueur de l'ordonnancement. Avec les notations précédentes, il revient à minimiser la date de fin d'exécution du dernier lot : B_m. Ce qui revient à minimiser la valeur : $s_m + C_{(m)}$.

Une dernière notation numérique :

$$\alpha = \frac{-1+\sqrt{5}}{2}$$

Finalement, une propriété sur le nombre α :

Propriété 5 $\alpha + \alpha^2 = 1$.

Démonstration :

$$\begin{aligned}
\alpha + \alpha^2 &= \alpha(1+\alpha) \\
&= \frac{\sqrt{5}-1}{2}\left(1+\frac{\sqrt{5}-1}{2}\right) = \left(\frac{\sqrt{5}-1}{2}\right)\left(\frac{\sqrt{5}+1}{2}\right) \\
&= \frac{1}{4}(\sqrt{5}-1)(\sqrt{5}+1) = \frac{1}{4}(5-1) \\
&= 1
\end{aligned}$$

□

11.3. Le problème hors-ligne

Pour le problème hors-ligne avec une machine à traitement par lot à capacité finie b, si toutes les tâches sont activées dans le système simultanément, la séquence optimale est construite en prenant d'abord les b tâches ayant le plus grand temps processeur puis en les ordonnançant dans un lot. Ensuite, nous réappliquons cette règle sur l'ensemble des tâches qu'il reste à exécuter jusqu'à ce qu'il ne reste plus aucune tâche à ordonnancer. Cet algorithme a pour complexité $\min\{O(n \log n), O(n^2/b)\}$ [BRU 98].

Si les tâches ne sont pas toutes activées à un même instant et si les lots sont toujours de capacité finie, alors le problème $1|p-batch, r_j, b = k|C_{max}$ est $\mathcal{NP}$-Difficile au sens fort ([BRU 98]). Il reste $\mathcal{NP}$-Difficile, mais au sens ordinaire s'il y a seulement deux dates d'activation différentes pour l'ensemble de toutes les tâches : $\{0, r\}$ [LIU 00]. Un algorithme pseudo-polynomial en temps est connu si l'ensemble des dates différentes d'activation des tâches est fini (*cf.* [LIU 00]). De plus, si les temps processeurs sont tous égaux, la minimisation de la longueur d'ordonnancement peut être résolue avec une complexité en temps polynomial [IKU 86].

Pour le problème général, $1 \mid p-batch, r_i, b = \infty \mid C_{max}$, un algorithme séparation et évaluation *branch and bound* et une heuristique dynamique ont été proposés dans [SUN 00] et un autre algorithme séparation et évaluation *branch and bound* a été énoncé dans [DUP 02].

Deux cas particuliers ont été étudiés :

– les tâches avec des échéances dans [CHE 01] ;

– les tâches avec contraintes de précédence dans [CHE 04].

Maintenant nous considérons le problème d'ordonnancement par une machine à traitement par lot de capacité infinie ou du moins suffisamment grande pour ordonnancer toutes les tâches dans un unique lot. Nous étudions la minimisation de la longueur d'ordonnancement. Le problème est dénoté, $1|p-batch, r_i, b = \infty|C_{max}$. Ce problème peut être résolu en temps polynomial $O(n^2)$ par la programmation dynamique [LEE 99] . Avec une structure de données plus évoluée, la complexité calculatoire peut être baissée à $O(n \log n)$ ([POO 00]).

Nous présentons maintenant l'algorithme défini dans [LEE 99] et basé sur une simple propriété de l'ordonnancement optimal.

Propriété 6 *Pour le problème $1 \mid p-batch, r_i, b = \infty \mid C_{max}$, s'il existe des tâches τ_i et τ_j telles que si $r_i < r_j$ et $C_i \leq C_j$, alors, il existe un ordonnancement optimal qui ordonnance la tâche τ_i dans le même lot que la tâche τ_j.*

Cette propriété est vraie dès lors que la tâche τ_i arrive avant la tâche τ_j alors elle ne retarde pas le lancement de l'exécution du lot. De plus, comme la durée d'exécution de la tâche τ_i est inférieure à celle de la tâche τ_j, l'inclure dans le lot n'allonge pas la durée d'exécution du lot.

Remarque 16 *D'après la propriété 6, l'algorithme proposé par [LEE 99] se concentre uniquement sur les tâches vérifiant :*

$$r_{i_1} < r_{i_2} < \ ... < r_{i_m}$$
$$C_{i_1} > C_{i_2} > ... > C_{i_m}$$

Pour les autres tâches de la configuration qui ne respectent pas cette propriété, l'algorithme choisit un lot pour ordonnancer chaque tâche en laissant inchangée la longueur d'ordonnancement.

Ainsi, en renommant les indices, nous n'étudions que les configurations de la forme suivante :

$$r_1 < r_2 < \ ... < r_m$$
$$C_1 > C_2 > ... > C_m$$

Soit $f(k)$, la valeur du C_{max} minimum pour la configuration partielle contenant les k premières tâches de la configuration, avec $f(\emptyset) = 0$, alors la solution optimale est donnée par $f(m)$.

$$f(k) = \min_{1 \leq i \leq k} \{\max\{f(k-i), r_k\} + C_{k-i+1}\} \tag{11.1}$$

Puis pour construire la séquence d'exécution, il faut repartir en sens inverse et commencer par le dernier lot. D'après la formule 11.1, il existe un i (avec $1 \leq i \leq m$) tel que les tâches $\tau_{m-i+1}, \ldots, \tau_m$ sont ordonnancées dans le dernier lot pour une durée de C_{m-i+1} ; pour la date de début d'exécution, il y a deux possibilités :

– $f(m) = r_m + C_{m-i+1}$, le dernier lot est ordonnancé à l'instant r_m ;

– $f(m) = f(m-i) + C_{m-i+1}$, le dernier lot commence juste après la fin d'exécution du lot précédent, à l'instant $f(m-i)$.

Dans les deux cas, il faut étudier la valeur de $f(m-i)$ pour connaître le reste de la séquence.

Exemple 18 *Exemple de l'application de l'algorithme optimal de* LEE *et* UZSOY *(formule 11.1) sur la configuration de tâches I représentée dans le tableau 11.1. L'algorithme optimal transforme la configuration pour appliquer la formule 11.1. Ainsi, de la configuration initiale, ne sont conservées que les tâches telles que si $r_i < r_j$ alors $C_i > C_j$. Après transformation, il ne reste que la tâche τ_5. Ainsi l'ordonnancement par l'algorithme optimal est de regrouper l'exécution de toutes les tâches dans un unique lot à l'activation de la tâche τ_5 et pendant une durée de C_5.*

Tâches	r_i	C_i
τ_1	0	2
τ_2	1	2
τ_3	2	3
τ_4	4	4
τ_5	5	4

Tableau 11.1. *Configuration de tâches, I, pour le problème d'ordonnancement d'une machine à traitement par lot.*

11.4. Le problème en-ligne

[LIU 00] présente un algorithme glouton (*greedy algorithm*) qui résout le problème d'ordonnancement par une machine à traitement par lot de capacité finie b. Cette règle est appelée H, et définie comme suit : *À n'importe quel instant, quand la machine est inoccupée et que certaines tâches sont disponibles, l'algorithme ordonnance les b tâches ayant le plus grand temps processeur dès que possible dans un lot*. Ces auteurs ont démontré que cet algorithme est 2-compétitif et que cette limite est aussi la borne inférieure pour les algorithmes conservatifs (c.-à-d. que cet algorithme fait partie des meilleurs algorithmes en-ligne pour ce problème).

Le ratio de compétitivité pour le problème d'ordonnancement par une machine à traitement par lot à capacité finie et pour la minimisation de la longueur d'ordonnancement est 2 ([LIU 00]). Mais dans [ZHA 01], les auteurs ont démontré que l'insertion de temps creux avant le début d'exécution des lots améliore la garantie de performance atteinte par les algorithmes en-ligne et la fixent à $(1+\sqrt{5})/2$. Ils fournissent également un algorithme en-ligne optimal pour le cas des machines à capacité infinie avec une garantie de performance de $(1+\sqrt{5})/2$. Ils conjecturent aussi sur un algorithme en-ligne optimal (avec une garantie de performance de $(1+\sqrt{5})/2$) pour le problème de machines à traitement par lot à capacité finie et pour les configurations de tâches ayant seulement deux dates d'activation différentes. Nous présentons maintenant l'algorithme en-ligne de DENG [DEN 99] et celui défini dans [ZHA 01] pour le problème des machines à capacité infinie ($1|p-batch, r_j, b=\infty|C_{max}$), appelé H^{∞}.

11.4.1. *Algorithme* DENG *[DEN 99]*

Ce paragraphe présente l'algorithme de DENG.

Exemple 19 *Nous présentons l'ordonnancement par l'algorithme de* DENG *de la configuration I. À l'instant 0, la tâche τ_1 arrive et le lancement du premier lot est alors retardé jusqu'à l'instant $(1+\alpha)r_1+\alpha C_1 = 1.23$. Durant cette attente, la tâche τ_2 arrive et retarde encore plus ce lancement jusqu'à l'instant $(1+\alpha)r_2+\alpha C_2 = 2.85$. La troisième tâche arrive entre-temps et retarde encore ce lancement jusqu'à l'instant $(1+\alpha)r_3+\alpha C_3 = 5.09$. Avec les nouvelles arrivées de τ_4 et τ_5, finalement, un seul lot est lancé contenant les cinq tâches*

Algorithme 2 : L'algorithme de DENG

Donnée : I, une configuration de tâches.
Donnée : $U(0)$, l'ensemble des tâches libre à l'instant 0.
Résultat : Ordonnancement de I par l'algorithme H^{∞}.
Début
  $t \longleftarrow 0$;
  Tant que *Vrai* **faire**
    $\gamma \longleftarrow \max_{\tau_j \in U(t)}(1+\alpha)r_j + \alpha C_j$;
    $s \longleftarrow \max\{t, \gamma\}$;
    Durant *l'intervalle de temps* $[t, s]$ **faire**
      Pour chaque *tâche* τ_h *qui arrive dans le système à l'instant* t' **faire**
        Si $(1+\alpha)r_h + \alpha C_h > \gamma$ **alors**
          $\gamma \longleftarrow (1+\alpha)r_h + \alpha C_h$;
          $t \longleftarrow t'$;
          $s \longleftarrow \max\{t, \gamma\}$;
      $U(t) \longleftarrow U(t) \cup \{\tau_h\}$;
    Ordonnancement dans un même lot de l'ensemble de toutes les tâches disponibles $U(s)$;
    Si *une tâche arrive pendant l'exécution du lot* **alors**
      $t \longleftarrow s + C_k$;
    sinon
      $t \longleftarrow r_h$, où τ_h est la prochaine tâche qui s'active;
fin

à l'instant $(1+\alpha)r_5 + \alpha C_5 = 10.56$ *et pour une durée de* 4 *unités de temps. La longueur d'ordonnancement est égale à* $(1+\alpha)r_5 + \alpha C_5 + 4 = 14.56$. *La figure 11.3 illustre l'ordonnancement de* I *construit par cet algorithme. Notons que l'algorithme obtient presque ici ses pires performances :* $\frac{14.56}{9} \backsimeq 1.617 \leq 1.618 \backsimeq \frac{1+\sqrt{5}}{2}$.

Theorème 19 *[DEN 99] L'algorithme de* DENG *fait partie des meilleurs algorithmes en-ligne pour le problème* $1 \mid p-batch, r_i, b = \infty \mid C_{max}$.

11.4.2. *L'algorithme en-ligne* H^{∞} *[ZHA 01]*

Nous allons maintenant présenter l'algorithme H^{∞} [ZHA 01]. Cet algorithme en-ligne est l'un des meilleurs algorithmes pour le problème d'ordonnancement étudié : $1 \mid p-batch, r_i, b = \infty \mid C_{max}$.

Remarque 17 *Le principe de cet algorithme est de toujours attendre l'instant* $(1+\alpha)r_k + \alpha C_k$, *avant de lancer l'exécution d'un lot, où* τ_k *est la tâche disponible, en attente d'être exécutée, avec le plus grand temps processeur requis. Ainsi, pour chaque lot* B_i, *le début de son exécution se fait à l'instant* $(1+\alpha)r_{(i)} + \alpha C_{(i)}$ *(cf. paragraphe 10.3).*

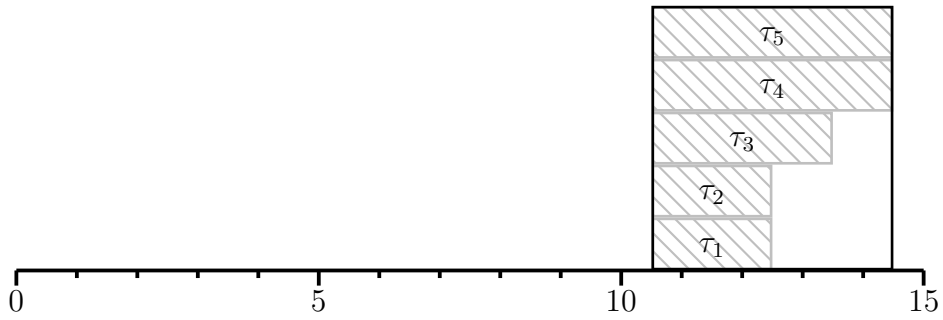

Figure 11.3. *Ordonnancement construit par l'algorithme de* DENG *pour l'ordonnancement de la configuration I présentée par le tableau 11.1.*

Algorithme 3 : H^{∞}

Donnée : I, une configuration de tâches.
Donnée : $U(0)$, l'ensemble des tâches libre à l'instant 0.
Résultat : Ordonnancement de I par l'algorithme H^{∞}.
Début
  $t \longleftarrow 0$;
  Tant que *Vrai* **faire**
    $k \longleftarrow h$ tel que $\tau_h \in U(t)$ et $C_h = \max\{C_j \mid \tau_j \in U(t)\}$;
    $\gamma \longleftarrow (1+\alpha)r_k + \alpha C_k$;
    $s \longleftarrow \max\{t, \gamma\}$;
    Durant *l'intervalle de temps* $[t, s]$ **faire**
      Pour chaque *tâche τ_h qui arrive dans le système à l'instant t'* **faire**
        Si $C_h > C_k$ **alors**
          $k \longleftarrow h$;
          $\gamma \longleftarrow (1+\alpha)r_h + \alpha C_h$;
          $t \longleftarrow t'$;
          $s \longleftarrow \max\{t, \gamma\}$;
      $U(t) \longleftarrow U(t) \cup \{\tau_h\}$;
    Ordonnancement dans un même lot de l'ensemble de toutes les tâches disponibles $U(s)$;
    Si *une tâche arrive pendant l'exécution du lot* **alors**
      $t \longleftarrow s + C_k$;
    sinon
      $t \longleftarrow r_h$, où τ_h est la prochaine tâche qui s'active;
fin

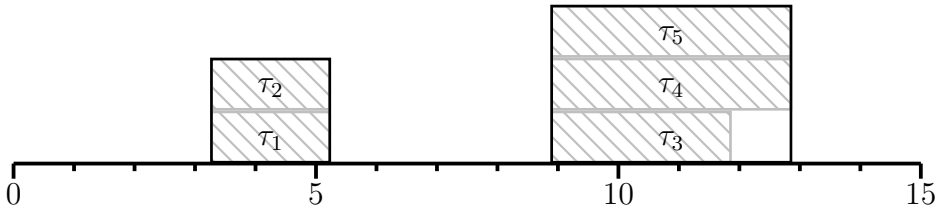

Figure 11.4. *Ordonnancement construit par H^∞ pour l'ordonnancement de la configuration I présentée par le tableau 11.1.*

Exemple 20 *Nous présentons l'ordonnancement de la configuration I (cf. tableau 11.1) par l'algorithme H^∞. À l'instant 0, la tâche τ_1 arrive et le premier lot est alors retardé jusqu'à l'instant $(1+\alpha)r_1+\alpha C_1 = 1.23$. Durant cette attente, la tâche τ_2 arrive mais avec un temps processeur pas plus important que celui de la tâche τ_1. Ainsi, le premier lot commence son exécution à l'instant 1.23, contenant les tâches τ_1 et τ_2 et est complété à l'instant 3.23. La tâche τ_3 arrive à l'instant 2, donc le second lot est retardé jusqu'à l'instant $(1+\alpha)r_3+\alpha C_3 = 5.09$. Durant ce délai d'attente, les tâches τ_4 et τ_5 arrivent et retardent encore le début d'exécution du lot jusqu'à l'instant $(1+\alpha)r_4+\alpha C_4 = 8.94$. À cet instant le second lot est ordonnancé, contenant les tâches τ_3, τ_4 et τ_5, et est complété à l'instant 12.94. La longueur d'ordonnancement est égale à $(1+\alpha)r_4+\alpha C_4+C_4 = 12.94$. La figure 11.4 présente l'ordonnancement de I construit par l'algorithme H^∞.*

Theorème 20 *[ZHA 01] L'algorithme de H^∞ fait partie des meilleurs algorithmes en-ligne pour le problème $1 \mid p-batch, r_i, b=\infty \mid C_{max}$.*

11.5. Conclusion

Dans ce chapitre, nous avons présenté plusieurs résultats en-ligne et hors-ligne pour l'ordonnancement de tâches par des machines à traitement par lot et pour la minimisation de la longueur d'ordonnancement. Notamment, pour le problème d'ordonnancement $1 \mid p-batch, r_i, b=\infty \mid C_{max}$, un algorithme hors-ligne optimal a été présenté. De plus, deux algorithmes en-ligne pour ce même problème ont été établis, mais ces algorithmes ont un délai d'attente qui dépend de la date d'arrivée des tâches ; par conséquent, selon le temps auquel une tâche survient, le délai d'attente peut s'avérer élevé.

11.6. Bibliographie

[BRU 98] BRUCKER P., GLADKY A., HOOGOVEEN H., KOVALYOV M., POTTS C., TAUTENHAHN T., VANDEVELDE S., « Scheduling a batching machine », *Journal of Scheduling*, vol. 1, n°1, p. 31-58, 1998.

[CHE 01] CHENG T., LIU Z., YU W., « Scheduling jobs with release dates and deadlines on a batch processing machines », *IIE Transactions*, vol. 33, p. 685-690, 2001.

[CHE 04] CHENG T., YAN J., YANG A., « Scheduling a batch-processing machine subject to precedence constraints, release dates and identical processing times », *Computer and Operations Research*, vol. 32, p. 849-859, 2004.

[DEN 99] DENG X., POON C. K., ZHANG Y., « Approximation Algorithms in Batch Processing », vol. LNCS 1741, p. 153-162, December 1999.

[DUP 02] DUPONT L., DHAENENS-FLIPO C., « Minimizing the makespan on a batch machine with non-identical job sizes : an exact procedure », *Computer and Operations Research*, vol. 29, n°7, p. 807-819, 2002.

[IKU 86] IKURA Y., GIMPLE M., « Efficient scheduling algorithms for a single batch processing machine », *Operations Research Letters*, vol. 5, p. 61-65, 1986.

[LEE 99] LEE C., UZSOY R., « Minimizing makespan on a single batch processing machine with dynamic job arrivals », *International Journal of Production Research*, vol. 37, n°1, p. 219-236, 1999.

[LIU 00] LIU Z., YU W., « Scheduling one batch processor subject to job release dates », *Discrete Applied Mathematics*, vol. 105, p. 129-136, 2000.

[POO 00] POON C., ZHANG P., « Minimizing makespan in Batch machine Scheduling », *in : proc Int. Symposium on Algorithms and Computations (ISAAC 2000), Lecture Notes in Computer Science, Springer Verlag*, , n°LNCS 1969, p. 386-397, 2000.

[SUN 00] SUNG C., CHOUNG Y., « Minimizing makespan on a single burn-in oven in semiconductor manufacturing », *European Journal of Oerational Research*, vol. 120, p. 559-574, 2000.

[ZHA 01] ZHANG G., CAI X., WONG C., « On-line algorithms for minimizing makespan on batch processing machines », *Naval Research Logistics*, vol. 48, p. 241-258, 2001.

Chapitre 12

Algorithmes d'ordonnancement pour les machines à traitement par lot

12.1. Introduction

Comme il l'a été démontré dans [ZHA 01], il est plus bénéfique (pour la minimisation de la longueur d'ordonnancement) d'attendre pour voir si de nouvelles tâches arrivent dans un futur proche dans le système avant de commencer l'exécution d'un lot. Par conséquent, les algorithmes que nous présentons dans ce chapitre reprennent ce principe et donc insèrent un intervalle de temps creux avant d'exécuter les lots.

Dans le paragraphe 12.2, nous développons une nouvelle démonstration de la borne inférieure du problème d'ordonnancement dans le cas général qui tient aussi lorsque les tâches sont de durée identique. Nous proposons ensuite trois nouveaux algorithmes : dans le paragraphe 12.3, nous présentons l'algorithme en-ligne αH ainsi que ses résultats positifs dans certains cas particuliers et sa garantie de performance dans le cas général. Dans le paragraphe suivant, paragraphe 12.4, l'algorithme $\alpha H2$ est développé ainsi que ses résultats dans certains cas particuliers et sa performance garantie pour le problème général : $1 \mid p - batch, r_i, b = \infty \mid C_{max}$. Finalement, le paragraphe 12.5 expose un troisième algorithme en-ligne qui obtient toujours une plus petite longueur d'ordonnancement que l'algorithme H^{∞} traité dans [ZHA 01] (*cf.* paragraphe 11.4.2), tout en obtenant le même ratio de compétitivité.

12.2. La borne inférieure

Dans [ZHA 01], les auteurs ont démontré que la borne inférieure du ratio de compétitivité pour les algorithmes en-ligne et pour le problème $1 \mid p-batch, r_i, b \mid C_{max}$ est de $(1+\sqrt{5})/2$. Leur démonstration utilise l'analyse de compétitivité avec un adversaire qui ne tient pas compte du choix de l'algorithme en-ligne pour définir les tâches à venir (*oblivious adversary*). Nous présentons une autre démonstration de cette borne inférieure avec un adversaire qui définit au fur et à mesure que l'algorithme en-ligne fait ses choix d'ordonnancement, la configuration de tâches pour mener l'algorithme en-ligne à sa pire performance

(*on-line adaptative adversary*). Cette nouvelle démonstration, à l'inverse de celle présentée dans [ZHA 01], reste valable dans le cas où tous les temps processeurs sont égaux.

Theorème 21 *La borne inférieure du ratio de compétitivité de n'importe quel algorithme en-ligne déterministe pour le problème d'ordonnancement* $1 \mid p-batch, r_i, b, p_i = p \mid C_{max}$*, est de* $\frac{1+\sqrt{5}}{2}$.

Démonstration :

Pour déterminer cette borne inférieure, nous utilisons l'analyse de compétitivité avec un adversaire évolutif. Ainsi, il construira au fur et à mesure la configuration de tâches qui mènera l'algorithme en-ligne à sa pire performance. L'algorithme en-ligne sera noté dans la suite de cette démonstration : A.

À l'instant 0, l'adversaire génère $b-1$ tâches arrivant dans le système avec un temps processeur de p (où p est un entier positif).

Notons par S, l'instant auquel l'algorithme A décide d'ordonnancer l'ensemble de ces $b-1$ tâches dans un unique lot d'exécution.

Pour compléter la configuration, l'adversaire évolutif a deux possibilités :

– Soit il décide de ne générer aucune nouvelle tâche. Dans ce cas, l'algorithme A a attendu inutilement jusqu'à l'instant S alors que l'algorithme hors-ligne a ordonnancé les $b-1$ tâches dès leur apparition, à l'instant 0. Nous obtenons alors, si nous notons σ_A la longueur d'ordonnancement obtenue par l'algorithme A et σ^* celle obtenue par l'adversaire, les longueurs d'ordonnancement suivantes :

$$\begin{aligned} \sigma_A &= S + p \\ \sigma^* &= p \end{aligned} \tag{12.1}$$

Calcul du ratio de compétitivité pour l'algorithme A :

$$c_A = \frac{S+p}{p} \tag{12.2}$$

– Soit l'algorithme évolutif génère une nouvelle tâche de temps processeur toujours égal à p et qui arrive dans le système à l'instant $S+\epsilon$ (où ϵ est un nombre positif, proche de zéro). Cette nouvelle tâche s'active juste après que l'algorithme en-ligne A ait commencé l'exécution du lot contenant les $b-1$ premières tâches. Ainsi deux lots lui sont nécessaires et le dernier lot démarre son exécution au mieux, à la fin du premier lot, c'est-à-dire à l'instant $S+p$. Quant à l'algorithme hors-ligne, il ordonnance toutes les tâches dans un même et unique lot à l'instant $S+\epsilon$. La longueur d'ordonnancement obtenue par chaque algorithme est de :

$$\begin{aligned} \sigma_A &= S + p + p \\ \sigma^* &= S + \epsilon + p \end{aligned}$$

Le calcul du ratio de compétitivité pour l'algorithme A, donne le résultat suivant :

$$c_A = \frac{S + 2p}{S + p + \epsilon} \tag{12.3}$$

Maintenant, pour calculer le ratio de compétitivité d'un algorithme en-ligne A, les formules 12.2 et 12.3 sont regroupées et pour considérer le pire comportement de notre algorithme A, nous considérons la formule qui mène au plus grand ratio :

$$c_A \geq \max\left(\frac{S + p}{p}; \frac{S + 2p}{S + p + \epsilon}\right) \tag{12.4}$$

La formule 12.4 est minimisée en posant :

$$\begin{cases} \epsilon = 0 \\ \frac{S+p}{p} = \frac{S+2p}{S+p+\epsilon} \end{cases}$$

En regroupant ce qui dépend de S, nous obtenons,

$$\begin{aligned} \frac{S + p}{p} &= \frac{S + 2p}{S + p} \\ (S + p)^2 &= p(S + 2p) \\ S(S + p) &= p^2 \end{aligned}$$

En considérant S l'inconnu, nous obtenons l'équation du second degré suivante :

$$S^2 + Sp - p^2 = 0$$

Le discriminant est égal à $5p^2 > 0$. Il y a donc deux solutions réelles : $S_1 = p\frac{-1+\sqrt{5}}{2}$ et $S_2 = p\frac{-1-\sqrt{5}}{2}$. Mais S_2 est une valeur négative et comme S représente un instant dans le temps, $S > 0$. Donc la solution de l'équation 12.5 est :

$$S = p\frac{-1 + \sqrt{5}}{2} \tag{12.5}$$

Il reste à reporter la valeur de S obtenue par la formule (12.5) dans l'expression (12.2). Nous obtenons alors,

$$\frac{S + p}{p} = \frac{p\frac{-1+\sqrt{5}}{2} + p}{p} = \frac{1 + \sqrt{5}}{2}$$

Finalement, en reprenant l'expression 12.4, on obtient que la borne inférieure du ratio de compétitivité de l'algorithme en-ligne A soit supérieure ou égale à $\frac{1+\sqrt{5}}{2}$.

□

12.3. Les problèmes particuliers : l'algorithme αH

Les algorithmes en-ligne αH et $\alpha H2$ pour chaque nouvelle tâche, τ, qui arrive dans le système, attendent pendant une durée βC où $0 \leq \beta \leq 1$. À noter que ces algorithmes étendent la définition de la règle H, paragraphe 11.4. Ces deux algorithmes définissent la règle H dans le cas particulier où le paramètre β tend vers 0.

Le principe de l'algorithme αH est le suivant : à n'importe quel instant, quand la machine est inoccupée (c.-à-d. qu'un lot n'est pas en cours d'exécution), et qu'il y a des tâches actives dans le système, encore non ordonnancées, alors pour chaque tâche τ_j non ordonnancée, le prochain lot ne doit pas être exécuté avant l'instant $r_j + \beta C_j$. Toutes les tâches disponibles sont alors ordonnancées dans un unique lot.

Algorithme 4 : αH

Donnée : I, une configuration de tâches.
Donnée : $U(0)$, l'ensemble des tâches libre à l'instant 0.
Résultat : Ordonnancement de I par l'algorithme αH.
Début
  $t \longleftarrow 0$;
  Tant que *Vrai* **faire**
    $\gamma \longleftarrow \max_{\tau_k \in U(t)}(r_k + \beta C_k)$;
    $s \longleftarrow \max\{t, \gamma\}$;
    Durant *l'intervalle de temps* $[t, s]$ **faire**
      Pour chaque *tâche* τ_h *qui arrive dans le système à l'instant* t' **faire**
        Si $r_h + \beta C_h > \gamma$ **alors**
          $\gamma \longleftarrow r_h + \beta C_h$;
          $t \longleftarrow t'$;
          $s \longleftarrow \max\{t, \gamma\}$;
      $U(t) \longleftarrow U(t) \cup \{\tau_h\}$;
    Ordonnancement dans un même lot de l'ensemble de toutes les tâches disponibles $U(s)$;
    Si *une tâche arrive pendant l'exécution du lot* **alors**
      $t \longleftarrow$ date de fin d'exécution du lot;
    sinon
      $t \longleftarrow r_h$, où τ_h est la prochaine tâche qui s'active;
fin

Exemple 21 *Nous présentons maintenant un exemple d'ordonnancement d'une configuration de tâches par l'algorithme αH avec $\beta = \alpha$. Nous utilisons la configuration I représentée dans le tableau 11.1 (page 153). Cet ordonnancement est illustré par la figure 12.1. La tâche τ_1 arrive dans le système à l'instant 0, ainsi le premier lot est retardé jusqu'à l'instant $r_1 + \alpha C_1 = 1.23$. La tâche τ_2 arrive pendant ce délai d'attente, à l'instant 1 et retarde l'ordonnancement du premier lot jusqu'à l'instant $r_2 + \alpha C_2 = 2.23$. La tâche τ_3 arrive*

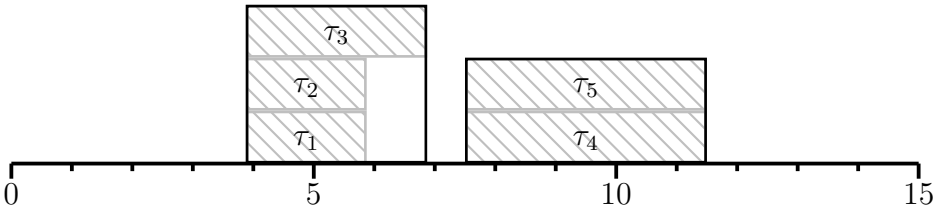

Figure 12.1. *Ordonnancement obtenu par l'algorithme αH pour la configuration définie dans le tableau 11.1.*

dans le système à l'instant 2 *et retarde à son tour l'ordonnancement du premier lot jusqu'à l'instant* $r_3 + \alpha C_3$*. Le premier lot est ordonnancé à cet instant (*$r_3 + \alpha C_3 = 3.85$*) et finit son exécution à l'instant* 6.85*. Les dernières tâches (*τ_4 *et* τ_5*) arrivent pendant l'exécution du premier lot. Le deuxième lot est donc exécuté à l'instant* $r_5 + \alpha C_5 = 7.47$ *et complété à l'instant* $7.47 + 4 = 11.47$*.*

Dans la suite, nous considérons $\beta = \alpha$ car c'est pour cette valeur que les résultats les plus intéressants sont obtenus.

Propriété 7 *Pour chaque lot régulier* B*, le début d'exécution est l'instant*

$$\max_{\tau_j \in B} (r_j + \alpha C_j)$$

12.3.1. *Temps processeurs égaux*

Dans ce paragraphe, nous démontrons que l'algorithme d'ordonnancement αH est l'un des meilleurs algorithmes en-ligne possibles pour la minimisation de la longueur d'ordonnancement lorsque tous les temps processeurs sont égaux, le problème $1|p - batch, r_i, p_i = p, b = \infty|C_{max}$. Ainsi, dans la suite de ce paragraphe, nous supposerons que tous les temps processeurs sont égaux à p. Une telle hypothèse simplifie les problèmes d'ordonnancement hors-ligne, du point de vue de la complexité (*cf.* [BAP 00]). Nous supposerons également, sans perte de généralité, que les tâches sont indexées par ordre croissant sur les dates d'activation :

$$i < j \Rightarrow r_i \leq r_j. \tag{12.6}$$

Avant de démontrer que l'algorithme αH est l'un des meilleurs algorithmes, nous présentons d'abord, trois résultats.

Lemme 4 *Pour n'importe quelle configuration I de n tâches du problème d'ordonnancement $1|p-batch, r_i, p_i = p, b = \infty|C_{max}$, la longueur optimale d'ordonnancement est égale à $\sigma^*(I) = r_n + p$.*

Démonstration :

Comme les temps processeurs sont tous égaux à p, toutes les tâches peuvent être ordonnancées dans un unique lot sans augmenter la longueur de l'ordonnancement. De plus, comme les tâches sont rangées par ordre croissant sur les dates d'arrivée, l'unique lot peut être ordonnancé dès l'instant r_n. Par conséquent, la longueur optimale d'ordonnancement, obtenue en ordonnançant toutes les tâches dans un unique lot à l'instant r_n, est égale à r_n+p.

□

Lemme 5 *Soit I une configuration de n tâches du problème $1|p - batch, r_i, p_i = p, b = \infty|C_{max}$, alors la tâche τ_n finit toujours l'ordonnancement construit par l'algorithme αH.*

Démonstration : Durant l'ordonnancement construit par l'algorithme αH, les tâches sont ordonnancées dans l'ordre croissant de leur arrivée. Lorsqu'un lot est exécuté, toutes les tâches disponibles sont ordonnancées dans le lot. Ainsi la tâche τ_n est ordonnancée dans le dernier lot construit par l'algorithme αH.

□

Lemme 6 *Pour n'importe quelle configuration de tâches I, du problème $1|r_i, p_i = p, b = \infty|C_{max}$, αH finit l'exécution de la tâche τ_n (à l'instant F_n à pas plus de $2p$ unités de temps après sa date d'activation (r_n) : $F_n - r_n \leq 2p$.*

Démonstration :

Pour démontrer ce lemme, nous devons nous intéresser à l'ordonnancement par αH de la tâche τ_n. Deux cas sont à considérer suivant que la tâche τ_n a été ordonnancée dans un lot régulier ou retardé :

– si la tâche τ_n a été ordonnancée dans un lot régulier par l'algorithme αH au pire à l'instant $r_n + \alpha p$ (délai d'attente de τ_n qui est la dernière tâche). Le lot contenant la tâche τ_n est complété alors avant l'instant $r_n + (1 + \alpha)p$. De plus, $\alpha < 1$ donc $F_n - r_n \leq 2p$.

– sinon, la tâche τ_n est ordonnancée dans un lot retardé. Le pire cas survient lorsque la tâche τ_n arrive juste après le début de l'exécution du lot qui la précède. Ce lot commence ainsi son exécution à l'instant $r_n - \epsilon$ (où ϵ est nombre positif tendant vers zéro , $\epsilon < 1$) et la finit à l'instant $r_n + p - \epsilon$. Finalement, le lot contenant la tâche τ_n termine son exécution à l'instant $r_n + 2p - \epsilon$. Si on fait tendre ϵ vers 0, nous obtenons que $F_n - r_n \leq 2p$.

Pour conclure, dans tous les cas, $F_n - r_n \leq 2p$.

□

Theorème 22 *L'algorithme αH est $1+\alpha$-compétitif pour le problème $1|p-batch, r_i, p_i = p, b = \infty|C_{max}$.*

Démonstration : Soit I une configuration de tâches du problème $1|p-batch, r_i, p_i = p, b = \infty|C_{max}$. Utilisons maintenant l'analyse de compétitivité.

D'après le lemme 4, la longueur d'ordonnancement optimale et donc obtenue par l'adversaire est de $\sigma^*(I) = r_n + p$.

Selon le lemme 5, la tâche τ_n est toujours ordonnancée dans le dernier lot. Notons par $\sigma_{\alpha H}(I)$, la longueur d'ordonnancement atteinte par l'algorithme αH. Deux cas sont à considérer :

– si le lot B dans lequel la tâche τ_n est ordonnancée, est régulier, alors le lot B est complété à l'instant :

$$\max_{\tau_j \in B}(r_j + (\alpha+1)p) = r_n + (\alpha+1)p$$

Par conséquent, $\sigma_{\alpha H}(I) \leq r_n + (\alpha+1)p$ et donc, le pire ratio de compétitivité vérifie :

$$c_{\alpha H} \leq \sup_{\forall I} \leq \frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} \leq \frac{r_n + (\alpha+1)p}{r_n + p} \leq (\alpha+1) \tag{12.7}$$

– si le lot B est un lot retardé, alors d'après le lemme 6, la tâche τ_n vérifie la propriété $F_n - r_n \leq 2p$. Or, comme la tâche τ_n finit l'ordonnancement (*cf.* lemme 5), $\sigma_{\alpha H}(I) \leq r_n + 2p$. Alors, le pire ratio de compétitivité est dans ce cas :

$$c_{\alpha H}\frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} \leq \frac{r_n + 2p}{r_n + p} \leq \frac{\alpha p + 2p}{\alpha p + p} = \frac{\alpha+2}{\alpha+1}$$

Mais comme B est retardé, il n'est pas le premier lot de la séquence d'exécution. Ainsi, $r_n \geq \alpha p$. Sinon, toutes les tâches seraient ordonnancées dans un unique lot selon la règle αH.

$$c_{\alpha H}\frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} \leq \frac{\alpha p + 2p}{\alpha p + p} = \frac{\alpha+2}{\alpha+1}$$

Si nous regroupons les deux cas, nous obtenons :

$$\frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} \leq \max\left(\alpha+1; \frac{\alpha+2}{\alpha+1}\right) \tag{12.8}$$

La pire valeur obtenue par l'expression 12.8 est atteinte quand :

$$\alpha + 1 = \frac{\alpha+2}{\alpha+1} \tag{12.9}$$

$$\alpha^2 + \alpha - 1 = 0 \tag{12.10}$$

Cette équation du second degré a un discriminant égal à 5, il est positif, donc il y a deux solutions réelles à cette équation, α devant être positif, $\alpha = (-1 + \sqrt{5})/2$. Ainsi, le ratio de compétitivité est de :

$$\frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} \leq \alpha + 1 = \frac{1+\sqrt{5}}{2} \approx 1.618 \tag{12.11}$$

□

12.3.2. *Temps processeurs agréables*

Nous démontrons dans ce paragraphe que si les temps processeurs sont agréables, alors l'algorithme d'ordonnancement αH est l'un des meilleurs algorithmes en-ligne. Commençons en premier lieu, par définir les tâches avec temps processeurs agréables.

Définition 25 *Une configuration de tâches a des temps processeurs agréables si et seulement si pour toutes tâches τ_i et τ_j de I :*

$$r_i < r_j \;\Rightarrow\; C_i \leq C_j$$

Avant de démontrer le résultat de compétitivité de l'algorithme αH, commençons par présenter quelques propriétés sur les lots ordonnancés par l'algorithme αH, quand les configurations de tâches ont des temps processeurs agréables :

Lemme 7 *Pour n'importe quelle configuration de tâches I, comme les temps processeurs sont agréables, pour toute tâche τ_i et τ_j telles que $i < j$ alors $r_i + \alpha C_i \leq r_j + \alpha C_j$*

Démonstration : la démonstration est immédiate car si $i < j$, $r_i \leq r_j$ et $C_i \leq C_j$.

□

Remarque 18 *D'après le lemme 7, le temps d'attente pour un lot correspond à celui engendré par la dernière tâche du lot (la tâche avec le plus grand indice).*

Remarque 19 *D'après la remarque 18 et la définition 25, le temps processeur d'un lot est celui de la dernière tâche du lot (la tâche avec le plus grand indice). En particulier, le temps processeur du dernier lot est égal à C_n.*

Theorème 23 *L'algorithme αH est $(1+\sqrt{5})/2$-compétitif pour la minimisation de la longueur d'ordonnancement sur une machine à traitement par lot de capacité infinie et en ordonnançant des configurations de tâches ayant des temps processeurs agréables.*

Démonstration :

Pour démontrer ce théorème, nous nous intéressons au dernier lot régulier. À noter qu'après ce dernier lot régulier, les lots qui suivent sont exécutés sans temps creux, les uns après les autres puisqu'ils sont tous retardés (nous parlons alors de bloc, définition 24, page 150). Nous utilisons l'analyse de compétitivité. Notons par $\sigma_{\alpha H}(I)$ (respectivement $\sigma^*(I)$) la longueur d'ordonnancement obtenue par l'algorithme αH (respectivement, l'algorithme optimal). Deux cas sont à considérer suivant la longueur du dernier bloc :

– si le dernier lot régulier est B_m, son ordonnancement a commencé à l'instant $r_n + \alpha C_n$ (d'après la remarque 18, page 166). Ainsi, la longueur d'exécution obtenue avec l'algorithme αH est de :

$$\sigma_{\alpha H}(I) = s_{(m)} + C_{(m)} = r_n + (\alpha+1)C_n$$

Pour l'adversaire optimal, d'après le lemme 4 (page 164), $\sigma^*(I) \geq r_n + C_n$. Ainsi, le ratio de compétitivité $c_{\alpha H}$ de l'algorithme αH est de :

$$\begin{aligned} c_{\alpha H} &\leq \frac{r_n + (\alpha+1)C_n}{r_n + C_n} \\ &\leq 1 + \frac{\alpha C_n}{r_n + C_n} \end{aligned}$$

Or, comme $C_n \leq C_n + r_n$, nous obtenons que $c_{\alpha H} \leq 1 + \alpha$.

– si le dernier lot régulier est B_l avec $l < m$. Soit τ_k la dernière tâche du lot B_l, alors B_l a commencé son exécution à l'instant $s_{(l)} = r_k + \alpha C_k$. Comme les lots suivants sont retardés, la longueur d'ordonnancement obtenue avec l'algorithme αH est de :

$$\begin{aligned} \sigma_{\alpha H}(I) &= s_{(l)} + C_{(l)} + \cdots + C_{(m)} \\ &= r_k + \alpha C_k + \sum_{i=l}^{m} C_{(i)} \end{aligned}$$

La longueur d'ordonnancement atteinte par l'algorithme optimal est : $\sigma^*(I) \geq r_n + C_n$ (*cf.* lemme 4, page 164), et le ratio de compétitivité résultant :

$$c_A \leq \frac{r_k + \alpha C_k + \sum_{i=l}^{m} C_{(i)}}{r_n + C_n}$$

Comme la tâche τ_n n'est pas ordonnancée dans le lot B_{m-1} $r_n \geq r_{(m-1)} + \alpha C_{(m-1)}$. En utilisant le même principe, par itération, nous obtenons que $r_{(m-1)} \geq r_{(m-2)} + \alpha C_{(m-2)}$ et donc :

$$r_n \geq r_{(m-2)} + \alpha C_{(m-2)} + \alpha C_{(m-1)}$$

Et finalement, nous obtenons que $r_n \geq r_k + \alpha \sum_{i=l}^{m-1} C_{(i)}$. Donc, pour l'algorithme optimal, sa performance peut s'écrire : $\sigma^*(I) \geq r_{(m-2)} + \alpha C_{(m-2)} + \alpha C_{(m-1)} + C_n$. Ainsi, pour le ratio de compétitivité :

$$c_A \leq \frac{r_k + \alpha C_k + \sum_{i=l}^{m} C_{(i)}}{r_n + \alpha \sum_{i=l}^{m-1} C_{(i)} + C_n}$$

De plus, $C_k \leq C_n$ car les temps processeurs sont agréables et par conséquent, nous obtenons :

$$c_A \leq \frac{r_k + \alpha C_n + \sum_{i=l}^{m} C_{(i)}}{r_k + \alpha \sum_{i=l}^{m-1} C_{(i)} + C_n}$$

or αr_k est positif et $C_{(m)} = C_n$ donc :

$$\leq \frac{r_k + (1+\alpha) C_n + \sum_{i=l}^{m-1} C_{(i)}}{r_k + \alpha \sum_{i=l}^{m-1} C_{(i)} + C_n} + \frac{\alpha r_k}{r_k + \alpha \sum_{i=l}^{m-1} C_{(i)} + C_n}$$

$$\leq \frac{(1+\alpha) r_k + (1+\alpha) C_n + \sum_{i=l}^{m-1} C_(i)}{r_k + \alpha \sum_{i=l}^{m-1} C_{(i)} + C_n}$$

Mais comme $\alpha = (-1+\sqrt{5})/2$ d'après la propriété 5 alors $\alpha(1+\alpha) = 1$:

$$\leq \frac{(1+\alpha) r_k + (1+\alpha) C_n + (1+\alpha)\alpha \sum_{i=l}^{m} C_{(i)}}{r_k + \alpha \sum_{i=l}^{m-1} C_{(i)} + C_n}$$

$$\leq (1+\alpha) \frac{r_k + \alpha \sum_{i=l}^{m-1} C_{(i)} + C_n}{r_k + \alpha \sum_{i=l}^{m-1} C_{(i)} + C_n}$$

$$\leq (1+\alpha)$$

□

12.3.3. *Deux dates d'activation distinctes*

Dans ce paragraphe, nous étudions le cas où pour chaque configuration de tâches, il y a uniquement deux dates d'activation distinctes mais nous relaxons les contraintes sur les durées d'exécution. Dans ce cas spécifique, nous montrons que l'algorithme d'ordonnancement αH fait partie des meilleurs algorithmes en-ligne possibles. La version hors-ligne de ce problème est résolue en temps polynomial quand les lots sont de capacité infinie et en temps pseudo-polynomial si la capacité des lots est finie [IKU 86]. Pour cette étude, nous pouvons nous concentrer sur seulement deux tâches : la tâche ayant le plus long temps processeur et étant arrivée à la première date d'activation et la tâche ayant le plus long temps processeur mais étant arrivée à la seconde date d'activation.

Theorème 24 *L'algorithme αH est $(1+\sqrt{5})/2$-compétitif pour la minimisation de la longueur d'ordonnancement sur une machine à traitement par lot de capacité infinie et en ordonnançant des configurations de tâches ayant seulement deux dates d'activation distinctes. Ce ratio est atteint si $\alpha = \frac{-1+\sqrt{5}}{2}$.*

Démonstration :

Notons τ_1 (respectivement τ_2) la tâche avec le plus long temps processeur des tâches arrivant à la première (respectivement seconde) date d'activation. De plus, nous supposons sans perte de généralité que la première date d'activation est 0. Ainsi, la tâche τ_1 arrive à l'instant 0.

Le pire scénario pour l'algorithme αH est lorsque la seconde date d'activation a lieu quand le premier lot vient à peine d'être lancé. Ainsi, le premier lot contenant la tâche τ_1 est ordonnancé à l'instant αC_1. La tâche τ_2 arrive à l'instant $\alpha C_1 + \epsilon$ (où ϵ est un nombre proche de 0 et tendant vers 0, $\epsilon > 0$). Le lot contenant la tâche τ_2 doit être ordonnancé après la fin d'exécution du premier lot. Ainsi, la longueur d'ordonnancement obtenue par l'algorithme αH et notée $\sigma_{\alpha H}(I)$ est égale à :

$$\sigma_{\alpha H}(I) = \alpha C_1 + C_1 + C_2$$

L'algorithme optimal (l'adversaire) a deux possibilités pour son ordonnancement selon les valeurs α, C_1 et C_2 :

– il ordonnance toutes les tâches dans un unique lot à l'instant $\alpha C_1 + \epsilon$. La longueur d'ordonnancement de l'adversaire, notée $\sigma^*(I)$ est alors de $\alpha C_1 + \max(C_1, C_2)$;

– il ordonnance les tâches en deux lots. Le premier pour l'ordonnancement de la tâche τ_1 (dès l'instant 0) et le second pour les autres tâches. Comme $\alpha C_1 < C_1$, à la fin du premier lot, le second lot est lancé aussitôt. La longueur d'ordonnancement est : $\sigma^*(I) = C_1 + C_2$.

Le premier cas correspond au cas où $C_1 \ll C_2$ et donc pour le second cas, $C_1 \gg C_2$. Si nous regroupons ces deux cas, nous obtenons :

$$\sigma^*(I) = \min(\alpha C_1 + \max(C_1, C_2), C_1 + C_2)$$

Le ratio de compétitivité de l'algorithme αH, noté $c_{\alpha H}$ est alors :

$$c_{\alpha H} \leq \frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} = \frac{(1+\alpha)C_1 + C_2}{\min(\alpha C_1 + \max(C_1, C_2), C_1 + C_2)} \tag{12.12}$$

Après quelques calculs numériques, l'expression précédente est minimisée lorsque $\alpha = \frac{-1+\sqrt{5}}{2}$. Nous remplaçons dans la formule 12.12, α par sa valeur, en notant de plus que ce ratio est maximisé pour $C_1 = C_2 = C$:

$$c_{\alpha H} \leq \frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} = \frac{(2 + \frac{-1+\sqrt{5}}{2})C}{(1 + \frac{-1+\sqrt{5}}{2})C}$$

$$c_{\alpha H} \leq \frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} = \frac{\frac{4-1+\sqrt{5}}{2}}{\frac{2-1+\sqrt{5}}{2}}$$

$$c_{\alpha H} \leq \frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} = \frac{3+\sqrt{5}}{1+\sqrt{5}}$$

$$c_{\alpha H} \leq \frac{\sigma_{\alpha H}(I)}{\sigma^*(I)} = \frac{1+\sqrt{5}}{2}$$

□

12.3.4. *Ratio de compétitivité d'*αH *pour le problème général*

Dans ce paragraphe, nous établissons les ratios de compétitivité pour le problème général où la capacité de la machine est infinie (ou du moins, capable d'ordonnancer toutes les tâches dans un unique lot). Et nous prouvons que l'algorithme αH n'est pas meilleur que 2-compétitif pour le problème d'ordonnancement, $1 \mid p - batch, r_i, b = \infty \mid C_{max}$.

Theorème 25 *L'algorithme d'ordonnancement* αH *n'est pas meilleur que* 2*-compétitif pour le problème général d'ordonnancement de tâches par une machine à traitement par lot de capacité infinie même lorsque* $\beta = \alpha$.

Démonstration :

Pour démontrer ce théorème, nous utilisons l'analyse de compétitivité avec un adversaire inconscient. L'adversaire définit la configuration I suivante :

$$\begin{aligned}
\tau_1 &= (r_1 = 0, C_1 = p),\\
\tau_2 &= (r_2 = \alpha p, C_2 = 1),\\
\tau_3 &= (r_3 = \alpha p + \alpha, C_3 = 1),\\
\cdots &= (\ldots, \ldots)\\
\tau_{\lfloor \alpha p \rfloor - 1} &= (r_{\lfloor \alpha p \rfloor - 1} = \alpha p + (\lfloor \alpha p \rfloor - 1)\alpha,\ C_{\lfloor \alpha p \rfloor - 1} = 1)\\
\tau_{\lfloor \alpha p \rfloor} &= (r_{\lfloor \alpha p \rfloor} = p,\ C_{\lfloor \alpha p \rfloor} = 1)
\end{aligned}$$

où p est un nombre très grand ($p > 1$).

– αH ordonnance les tâches $\{\tau_1, \ldots, \tau_{\lfloor \alpha p \rfloor - 1}\}$ dans un premier lot. En effet, à chaque délai d'attente expiré, une nouvelle tâche arrive et retarde l'exécution du lot. À la fin de l'exécution du premier lot, la tâche $\tau_{\lfloor \alpha p \rfloor}$ a été activée dans le système et son délai d'attente est révolu, elle est alors ordonnancée seule dans un second lot : $B_1 = \{\tau_1, ..., \tau_{\lfloor \alpha p \rfloor - 1}\}$ et $B_2 = \{\tau_{\lfloor \alpha p \rfloor}\}$.

– l'algorithme optimal [LEE 99] ordonnance τ_1 dans un premier lot puis il regroupe toutes les autres tâches $\{\tau_2, \ldots, \tau_{\lfloor \alpha p \rfloor}\}$ dans un second lot : $B_1 = \{\tau_1\}$ et $B_2 = \{\tau_2, ..., \tau_{\lfloor \alpha p \rfloor}\}$.

Calculons maintenant les longueurs d'ordonnancement atteintes par ces deux algorithmes. Notons par $\sigma^*(I)$ et $\sigma_{\alpha H}(I)$, les longueurs d'ordonnancement atteintes respectivement par l'adversaire optimal et par l'algorithme αH. L'algorithme optimal, commence l'exécution de son dernier lot à l'instant p et pour une durée de 1, sa longueur d'ordonnancement est donc égale à $1+p$. αH a commencé l'exécution du dernier lot, en retard, à l'instant $\alpha p + \lfloor \alpha p \rfloor \alpha + p$ et pour une durée de 1, sa longueur d'ordonnancement est égale à $\alpha p + \lfloor \alpha p \rfloor \alpha + p + 1$. Ainsi, le ratio de compétitivité de l'algorithme αH vaut :

$$\begin{aligned}
c_{\alpha H} &= \lim_{p \to \infty} \frac{\alpha p + \lfloor \alpha p \rfloor \alpha + p + 1}{p + 1}\\
&\approx \frac{(\alpha^2 + \alpha)p + p + 1}{p + 1}
\end{aligned}$$

En utilisant la propriété 5, $(\alpha^2 + \alpha) = 1$ et

$$\begin{aligned} c_{\alpha H} &\approx \frac{2p+1}{p+1} \\ &\approx 2 \end{aligned}$$

□

12.3.5. *Conclusion*

Nous venons de présenter l'algorithme en-ligne αH. Cet algorithme fait partie des meilleurs algorithmes en-ligne dans le cas où tous les temps processeurs sont égaux, ou agréables ou finalement dans le cas où il n'y a uniquement que deux dates d'activation pour une configuration. Mais pour le problème général, cet algorithme ne fait pas partie des meilleurs algorithmes en-ligne, sa garantie de performance est de 2.

12.4. Les problèmes particuliers : l'algorithme $\alpha H2$

Cet algorithme est une variante de l'algorithme αH (*cf.* paragraphe 12.3) qui insère généralement moins de temps creux (délais d'attente plus courts) dans l'ordonnancement et obtient des garanties de performances identiques dans de nombreux cas.

Remarque 20 *De même que pour l'algorithme αH, dans la suite nous considérons $\beta = \alpha$.*

Exemple 22 *Cet exemple présente l'ordonnancement de la configuration de tâches I, représentée dans le tableau 11.1 (page 153), par l'algorithme $\alpha H2$. La première tâche τ_1 arrive à l'instant 0, l'exécution du premier lot est retardé jusqu'à l'instant $r_1 + \alpha C_1 = 1.23$. La tâche τ_2 arrive pendant ce laps de temps et est ordonnancée avec τ_1 dans le premier lot à l'instant $r_1 + \alpha C_1$. Le premier lot est complété au temps $r_1 + (1 + \alpha C_1) = 1.23 + 2 = 3.23$. Le second lot est ordonnancé à l'instant $r_3 + \alpha C_3 = 3.85$ en contenant uniquement τ_3 et est complété à l'instant $r_3 + (1 + \alpha)C_3 = 6.85$. Les dernières tâches arrivent pendant l'exécution du second lot et attendent la fin d'exécution de celui-ci pour être ordonnancées car leur délai d'attente est fini avant. Par conséquent, le troisième et dernier lot (contenant les tâches τ_4 et τ_5) est ordonnancé juste après la fin d'exécution du second lot à l'instant 6.85. La date de fin d'exécution de la configuration est égale à $6.85 + 4 = 10.85$. Cet ordonnancement est représenté par la figure 12.2.*

Propriété 8 *Pour chaque lot régulier B exécuté par l'algorithme $\alpha H2$, l'exécution du lot commence à l'instant $\min_{\tau_j \in B} (r_j + \alpha C_j)$.*

Algorithme 5 : $\alpha H2$

Donnée : I, une configuration de tâches.
Donnée : $U(0)$, l'ensemble des tâches libre à l'instant 0.
Résultat : Ordonnancement de I par l'algorithme $\alpha H2$.
Début
  $t \longleftarrow 0$;
  Tant que *Vrai* **faire**
    $\gamma \longleftarrow \min_{\tau_k \in U(t)}(r_k + \beta C_k)$;
    $s \longleftarrow \max\{t, \gamma\}$;
    Durant *l'intervalle de temps* $[t, s]$ **faire**
      Pour chaque *tâche* τ_h *qui arrive dans le système à l'instant* t' **faire**
        Si $r_h + \beta C_h < \gamma$ **alors**
          $\gamma \longleftarrow r_h + \beta C_h$;
          $t \longleftarrow t'$;
          $s \longleftarrow \max\{t, \gamma\}$;
      $U(t) \longleftarrow U(t) \cup \{\tau_h\}$;
    Ordonnancement dans un même lot de l'ensemble de toutes les tâches disponibles $U(s)$;
    Si *une tâche arrive pendant l'exécution du lot* **alors**
      $t \longleftarrow$ date de fin d'exécution du lot;
    sinon
      $t \longleftarrow r_h$, où τ_h est la prochaine tâche qui s'active;
fin

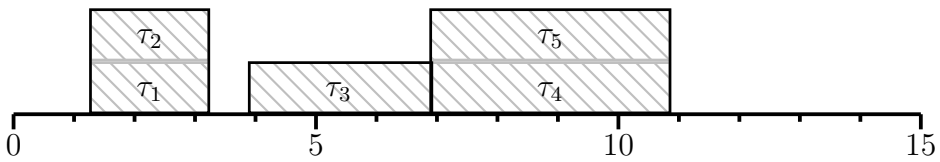

Figure 12.2. *Ordonnancement obtenu par l'algorithme $\alpha H2$ pour la configuration définie dans le tableau 11.1.*

12.4.1. *Temps processeurs égaux*

Nous commençons par établir quelques résultats.

Le lemme 5 (page 164) reste toujours valide pour l'algorithme d'ordonnancement $\alpha H2$. Ainsi, $\alpha H2$ ordonnance également les tâches par ordre de leur arrivée. De plus le lemme 6 (page 164) valable uniquement pour la dernière tâche avec l'algorithme αH peut s'étendre à toutes les tâches avec l'algorithme $\alpha H2$.

Lemme 8 *Pour n'importe quelle configuration I de tâches du problème $1|r_i, p_i = p, b = \infty|C_{max}$, l'algorithme $\alpha H2$ finit l'exécution de toutes les tâches moins de $2p$ unités de temps après leur arrivée dans le système : $F_k - r_k \leq 2p$.*

Démonstration :

Soit τ_k, une tâche de la configuration I. Comme lors de la démonstration du lemme 6 (page 164), deux cas sont à étudier :

– si la tâche τ_k est ordonnancée dans un lot régulier par l'algorithme $\alpha H2$ au pire à l'instant $r_k + \alpha p$ (délai d'attente de la tâche τ_k). Le lot contenant la tâche τ_k est complété alors avant l'instant $r_k + (1 + \alpha)p$. De plus, $\alpha < 1$ donc $F_k - r_k \leq 2p$.

– sinon, la tâche τ_k est ordonnancée dans un lot retardé. Le pire cas survient lorsque la tâche τ_k arrive juste après le début de l'exécution du lot qui la précède. Ce lot commence ainsi son exécution à l'instant $r_k - \epsilon$ (où ϵ est nombre positif tendant vers zéro , $\epsilon < 1$) et la finit à l'instant $r_k + p - \epsilon$. Finalement, le lot contenant la tâche τ_k termine son exécution à l'instant $r_k + 2p - \epsilon$. Si on fait tendre ϵ vers 0, nous obtenons que $F_k - r_k \leq 2p$.

Dans les ceux cas, le lemme est démontré.

□

Nous pouvons maintenant établir le résultat de compétitivité pour l'algorithme $\alpha H2$:

Theorème 26 *L'algorithme d'ordonnancement $\alpha H2$ est $(1 + \sqrt{5})/2$-compétitif pour le problème $1|p - batch, r_i, p_i = p, b = \infty|C_{max}$.*

Démonstration :

D'après le lemme 4 (page 164), la longueur d'ordonnancement obtenue par l'adversaire est égale à $r_n + p$.

Deux cas sont à étudier à propos du dernier lot :

– le dernier lot, B_m, est régulier : alors le dernier lot commence son exécution au pire à l'instant $r_n + \alpha p$. Ainsi, une borne supérieure de sa longueur d'ordonnancement est de $r_n + (1 + \alpha)p$. Par conséquent, nous obtenons un ratio de compétitivité $c_{\alpha H2}$ de l'algorithme $\alpha H2$ égal à :

$$c_{\alpha H2} \leq \frac{r_n + (1 + \alpha)p}{r_n + p} \leq (1 + \alpha)$$

– si le dernier lot n'est pas régulier (lot retardé), nous savons que toutes les tâches τ_k, d'après le lemme 8, finissent leur exécution moins de $2p$ unités de temps après leur activation. Ainsi, la longueur d'ordonnancement de l'algorithme $\alpha H2$ est limitée par $r_n \geq \alpha p$. Sinon,

la tâche τ_n est ordonnancée dans le premier lot qui est nécessairement régulier. Le ratio de compétitivité est alors de :

$$c_{\alpha H2} \leq \frac{r_n + 2p}{r_n + p} \leq \frac{\alpha p + 2p}{\alpha p + p} = \frac{\alpha + 2}{\alpha + 1}$$

En réutilisant la fin de la démonstration du théorème 22, nous obtenons que le ratio de compétitivité de l'algorithme $\alpha H2$ est de $(1 + \sqrt{5})/2$.

Pour conclure, le ratio de compétitivité de l'algorithme d'ordonnancement $\alpha H2$ est de $(1 + \sqrt{5})/2$ si $\alpha = (-1 + \sqrt{5})/2$

□

Remarque 21 *Dès l'instant où les algorithmes αH et $\alpha H2$ font partie des meilleurs algorithmes en-ligne, tout algorithme en-ligne qui choisit pour l'ensemble des tâches disponibles ($U(t)$ à l'instant t), d'attendre jusqu'à un instant compris entre $\min_{\tau_k \in U(t)}(r_k + \alpha C_k)$ et $\max_{\tau_k \in U(t)}(r_k + \alpha C_k)$ est aussi un des meilleurs algorithmes en-ligne pour le même problème.*

12.4.2. *Temps processeurs agréables*

Contrairement à l'algorithme αH, $\alpha H2$ n'est pas un des meilleurs algorithmes possibles lorsque les durées d'exécution sont agréables.

Theorème 27 *L'algorithme $\alpha H2$ n'est pas meilleur que 2-compétitif lors de l'ordonnancement par une machine à traitement par lot de capacité infinie et pour la minimisation de la longueur d'ordonnancement lorsque les temps processeurs d'une configuration sont agréables.*

Démonstration :

Nous utilisons l'analyse de compétitivité et étudions la configuration I à temps processeurs agréables suivante :

$$\tau_1 = (r_1 = 0, C_1 = 1),$$
$$\tau_2 = (r_2 = 0, C_2 = p),$$
$$\tau_3 = (r_3 = \alpha + \epsilon, C_3 = p)$$

où p est un entier très grand, $t > 1$ et ϵ un nombre inférieur à 1 et proche de zéro mais positif.

– $\alpha H2$ ordonnance les tâches τ_1 et τ_2 dans un premier lot, à l'instant α et pour une durée de p. Un second lot est ordonnancé juste après la fin d'exécution du premier et contenant uniquement τ_3. La longueur d'ordonnancement, notée $\sigma_{\alpha H2}(I)$ est égale à $\alpha + 2p$.

– l'algorithme optimal [LEE 99] ordonnance toutes les tâches dans un unique lot, à l'instant α, en faisant tendre ϵ vers 0. Sa longueur d'ordonnancement est égale à $\sigma^*(I) = \alpha + p$.

Calculons à présent le ratio de compétitivité de l'algorithme $\alpha H2$:

$$c_{\alpha H2} = \lim_{p \to \infty} \frac{\alpha + 2p}{\alpha + p} = 2$$

□

Remarque 22 *L'algorithme $\alpha H2$ a une garantie de performance de 2 lorsque les temps processeurs sont agréables, donc pour le problème général, la garantie de performance n'est pas meilleure que 2.*

12.4.3. *Deux dates d'activation distinctes*

Nous démontrons dans ce paragraphe que l'algorithme $\alpha H2$ n'est pas mieux que 2-compétitif lorsqu'il n'y a pour chaque configuration que deux dates d'activation distinctes. Nous levons les contraintes sur les durées d'exécution.

Theorème 28 *L'algorithme αH est 2-compétitif pour la minimisation de la longueur d'ordonnancement sur une machine à traitement par lot de capacité infinie et en ordonnançant des configurations de tâches ayant seulement deux dates d'activation distinctes.*

Démonstration :

Pour démontrer ce théorème, nous utilisons l'analyse de compétitivité et étudions la configuration de tâches I suivante :

$$\tau_1 = (r_1 = 0, C_1 = 1/p),$$
$$\tau_2 = (r_2 = 0, C_2 = p),$$
$$\tau_2 = (r_3 = \frac{1+\alpha}{p}, C_3 = p),$$

Où p est un entier positif strictement supérieur à 1.

– L'algorithme $\alpha H2$ attend jusqu'à l'instant $\frac{\alpha}{p}$ pour commencer l'exécution du premier contenant les tâches τ_1 et τ_2 (la tâche τ_3 n'étant pas encore activée). Ce lot se termine à l'instant $\frac{\alpha}{p} + p$. La fin d'exécution de ce lot est immédiatement suivie par l'exécution du second lot contenant la tâche τ_3 dont le délai d'attente s'est expiré à l'instant $\frac{1+\alpha}{p} + \alpha p$. Ainsi, sa longueur d'exécution, notée $\sigma_{\alpha H2}$, est égale à $\frac{\alpha}{p} + 2p$.

– L'algorithme optimal attend l'activation de la tâche τ_3 pour ordonnancer un unique lot contenant toutes les tâches. Sa longueur d'ordonnancement est notée σ^* et est égale à $\frac{\alpha+1}{p} + p$.

Ainsi, le ratio de compétitivité, dénoté $c_{\alpha H2}$ de l'algorithme $\alpha H2$ est égal à :

$$\begin{aligned} c_{\alpha H2} &= \frac{\frac{\alpha}{p} + 2p}{\frac{\alpha+1}{p} + p} \\ &= \frac{2p}{p} \frac{1 + \frac{\alpha}{2p^2}}{1 + \frac{\alpha+1}{p^2}} \end{aligned}$$

En faisant tendre p vers l'infini, nous obtenons :

$$c_{\alpha H2} = 2 \lim_{p \to \infty} \frac{1 + \frac{\alpha}{2p^2}}{1 + \frac{\alpha+1}{p^2}}$$

$$c_{\alpha H2} = 2$$

Ce qui conclut cette démonstration.

□

12.5. Le problème général : l'algorithme αH^∞

Nous avons présenté dans le paragraphe 11.4.2, [ZHA 01], l'algorithme en-ligne H^∞ qui est l'un des meilleurs algorithmes en-ligne pour le problème général avec des machines à capacité infinie. Mais cet algorithme insère de larges temps creux dans l'ordonnancement qu'il construit (avant le lancement de l'exécution d'un lot). En effet, si à un instant t, la tâche τ_k est celle disponible avec le plus grand temps processeur, le délai d'attente avant le lancement du lot est égal à $\alpha r_k + \alpha p_k$. L'insertion de ces larges temps creux a pour conséquence de ne jamais y avoir plus de deux lots ordonnancés à la suite sans temps creux et ce, même pour des configurations comprenant beaucoup de tâches (cette propriété est prouvée dans [ZHA 01]). En pratique, le temps creux inséré dans la séquence d'ordonnancement dépend du temps processeur et de la date d'arrivée de la tâche. Par conséquent, deux tâches avec le même temps processeur n'attendront pas pendant la même durée. Notre algorithme insère généralement moins de temps creux que l'algorithme H^∞, car le délai d'attente ne dépend que du temps processeur de la tâche. Ainsi, le nouvel algorithme que nous présentons, αH^∞, est basé sur le même principe que celui de [ZHA 01], H^∞. Mais nous supprimons le terme αr_k dans le calcul du délai d'attente. Nous démontrons, après l'avoir présenté, que cet algorithme insère moins de temps creux tout en obtenant la même garantie de performance par rapport au problème étudié.

Exemple 23 *Ordonnancement de la configuration I, représentée dans le tableau 11.1 (page 153) par l'algorithme αH^∞. La tâche τ_1 arrive dès l'instant 0 et introduit un délai d'attente jusqu'à l'instant $\alpha C_1 = 2\alpha = 1.23$. La tâche τ_2 arrive à l'instant 1 avec le même temps processeur que τ_1. Ainsi, elle ne retarde pas plus le début d'exécution du premier lot. Ce premier lot est exécuté à l'instant 1.23, contenant les tâches τ_1 et τ_2. Il finit son exécution à l'instant 3.23. La tâche τ_3 s'est activée pendant l'exécution du premier lot, à l'instant 2 et*

Algorithme 6 : αH^{∞}

Donnée : I, une configuration de tâches.
Donnée : $U(0)$, l'ensemble des tâches libre à l'instant 0.
Résultat : Ordonnancement de I par l'algorithme H^{∞}.
Début
  $t \longleftarrow 0$;
  Tant que *Vrai* **faire**
    $k \longleftarrow h$ tel que $\tau_h \in U(t)$ et $C_h = \max\{C_j \mid \tau_j \in U(t)\}$;
    $\gamma \longleftarrow r_k + \alpha C_k$;
    $s \longleftarrow \max\{t, \gamma\}$;
    Durant *l'intervalle de temps* $[t, s]$ **faire**
      Pour chaque *tâche* τ_h *qui arrive dans le système à l'instant* t' **faire**
        Si $C_h > C_k$ **alors**
          $k \longleftarrow h$;
          $\gamma \longleftarrow r_h + \alpha C_h$;
          $t \longleftarrow t'$;
          $s \longleftarrow \max\{t, \gamma\}$;
      $U(t) \longleftarrow U(t) \cup \{\tau_h\}$;
    Ordonnancement dans un même lot de l'ensemble de toutes les tâches disponibles $U(s)$;
    Si *une tâche arrive pendant l'exécution du lot* **alors**
      $t \longleftarrow s + C_k$;
    sinon
      $t \longleftarrow r_h$, où τ_h est la prochaine tâche qui s'active;
fin

est mise en attente jusqu'à l'instant $r_3 + \alpha C_3 = 3.85$*. Comme aucune autre tâche ne s'est réveillée pendant cet intervalle de temps, le second lot contenant uniquement la tâche* τ_3 *est ordonnancé à l'instant* 3.85 *et complété à l'instant* 6.85*. Les tâches* τ_4 *et* τ_5 *sont arrivées pendant l'exécution du second lot. Elles ont des temps processeurs égaux, le lot les exécutant devait attendre jusqu'à l'instant* $r_4 + \alpha C_4 = 6.47$ *pour commencer son exécution. Mais comme le processeur était toujours occupé, le lot a pu commencer son exécution juste après l'exécution du second lot, à l'instant* 6.85 *et l'a fini à l'instant* 10.85*. Cet ordonnancement est représenté par la figure 12.3.*

Propriété 9 *Soit* τ_k *la tâche ayant le plus grand temps processeur ordonnancé dans un lot régulier* B*, alors l'algorithme* αH^{∞} *a démarré* B *à l'instant* $r_k + \alpha C_k$*.*

Nous démontrons maintenant que l'algorithme αH^{∞} fait partie des meilleurs algorithmes en-ligne pour notre problème d'ordonnancement. Cette démonstration est malheureusement longue, mais nous n'avons pas trouvé le moyen de la réduire.

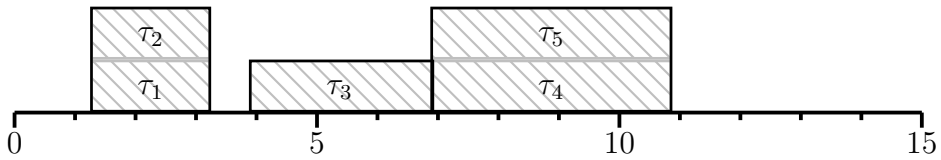

Figure 12.3. *Ordonnancement construit par l'algorithme αH^∞ pour la configuration I, tableau 11.1.*

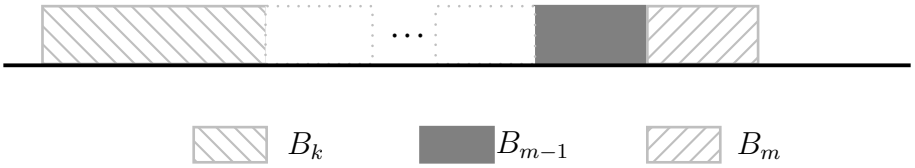

Figure 12.4. *L'algorithme αH^∞ ordonnance dans un seul bloc plusieurs lots.*

Theorème 29 *L'algorithme αH^∞ est $(1+\sqrt{5})/2$-compétitif pour le problème d'ordonnancement $1|p-batch, r_i, b=\infty|C_{max}$.*

Démonstration :

Pour démontrer ce théorème, nous utilisons l'analyse de compétitivité.

Attendu que αH^∞ est basé sur le même principe que l'algorithme H^∞ et qu'il introduit moins de temps creux dans l'ordonnancement, nous pouvons réutiliser, partiellement, les résultats obtenus avec l'algorithme H^∞. En effet, il a été prouvé [ZHA 01] que pour l'algorithme H^∞, le dernier bloc est composé d'au plus deux lots. Ainsi, pour notre démonstration, en réutilisant ces travaux, si la séquence d'exécution construite par l'algorithme αH^∞ se finit par un bloc d'au plus deux lots alors l'algorithme αH^∞ obtient un ratio de compétitivité de $(1+\sqrt{5})/2$.

Dans la suite, nous considérerons que dans la séquence d'ordonnancement construite par l'algorithme αH^∞, le dernier bloc est composé d'au moins trois blocs.
Il s'écrit $(B_k, \ldots, B_{m-1}, B_m)$ (où B_k est le dernier lot régulier de la séquence d'ordonnancement, comme le représente la figure 12.4).

Vu que le lot B_k est régulier, il commence son exécution à l'instant $s_{(k)} = r_{(k)} + \alpha C_{(k)}$, où $\tau_{(k)}$ désigne la tâche avec le plus long temps processeur du lot B_k. Ainsi, comme les lots suivants B_k sont retardés, la longueur d'ordonnancement obtenue par l'algorithme αH^∞,

notée $\alpha H^{\infty}(I)$ est de :

$$\alpha H^{\infty}(I) = r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m} C_{(i)}$$

Dans le bloc (représenté par la figure 12.4), seuls les lots B_k, B_{m-1} et B_m nous intéressent. Les autres lots, les lots $B_{k+1}, \ldots, B_{m-2}$, s'ils existent, n'interviennent que dans la somme en tant que valeur numérique. Trois cas sont à étudier suivant les durées d'exécution des lots B_k, B_{m-1} et B_m :

$$\begin{aligned} C_{(k)} &\geq C_{(m-1)} \\ C_{(m)} &\geq C_{(k)} \; et \; C_{(m-1)} \leq C_{(k)} \\ C_{(m-1)} &\geq C_{(k)} \geq C_{(m)} \end{aligned}$$

Nous distinguons les trois cas et les traitons séparément :

– $C_{(k)} \geq C_{(m-1)}$:

En utilisant le résultat du lemme 4 (page 164), l'adversaire optimal doit ordonnancer la tâche $\tau_{(m)}$ donc si nous notons $\sigma^*(I)$, la longueur d'ordonnancement obtenue par l'algorithme optimal, $\sigma^*(I) \geq r_{(m)} + C_{(m)}$.
De plus, la tâche $\tau_{(m)}$ arrive après le début de l'exécution du lot B_{m-1} par l'algorithme αH^{∞} car sinon, elle serait exécutée dans le lot B_{m-1}. Mais le lot B_k termine son exécution par l'algorithme αH^{∞} à l'instant $r_{(k)} + \alpha C_{(k)} + C_{(k)}$ et donc comme tous les lots qui suivent sont retardés, le lot B_{m-1} commence son exécution juste après la fin d'exécution du lot B_{m-2}, à l'instant $r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)}$. Ainsi,

$$\begin{aligned} r_{(m)} &\geq r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} \\ \sigma^*(I) &\geq r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)} \end{aligned}$$

Le ratio de compétitivité vaut :

$$\begin{aligned} c_{\alpha H^{\infty}} &\leq \frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m} C_{(i)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}} \\ c_{\alpha H^{\infty}} &\leq \frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}} + \frac{C_{(m-1)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}} \\ c_{\alpha H^{\infty}} &\leq 1 + \frac{C_{(m-1)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}} \end{aligned}$$

Mais en utilisant la propriété 5, $\alpha + \alpha^2 = 1$, l'inégalité précédente peut être simplifiée :

$$\begin{aligned} c_{\alpha H^\infty} &\leq 1 + \frac{C_{(m-1)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}} \\ c_{\alpha H^\infty} &\leq 1 + \alpha \frac{(1+\alpha) C_{(m-1)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}} \end{aligned}$$

Or, d'après notre première hypothèse, $C_{(k)} \geq C_{(m-1)}$:

$$\begin{aligned} c_{\alpha H^\infty} &\leq 1 + \alpha \frac{(1+\alpha) C_{(k)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}} \\ c_{\alpha H^\infty} &\leq 1 + \alpha \frac{(1+\alpha) C_{(k)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + C_{(k)} + C_{(m)}} \\ c_{\alpha H^\infty} &\leq 1 + \alpha \end{aligned}$$

– $C_{(m)} \geq C_{(k)}$:

Comme dans le cas précédent,

$$\sigma^*(I) \geq r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}$$

Le ratio de compétitivité est alors de :

$$c_{\alpha H^\infty} \leq \frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m} C_{(i)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}}$$

Mais $\alpha r_{(k)} \geq 0$ et $\alpha \sum_{i=k+1}^{m-2} C_{(i)} \geq 0$. Par conséquent, sans perte de généralité, nous pouvons ajouter ces éléments au numérateur de ce ratio :

$$\begin{aligned} c_{\alpha H^\infty} &\leq \frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m} C_{(i)} + (\alpha r_{(k)} + \alpha \sum_{i=k+1}^{m-2} C_{(i)})}{r_{(k)} + \alpha C_{(k)+} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}} \\ c_{\alpha H^\infty} &\leq \frac{(1+\alpha) r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m} C_{(i)} + \alpha \sum_{i=k+1}^{m-2} C_{(i)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}} \\ c_{\alpha H^\infty} &\leq \frac{(1+\alpha) r_{(k)} + \alpha C_{(k)} + C_{(k)} + (1+\alpha) \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m-1)} + C_{(m)}}{r_{(k)} + \alpha C_{(k)} + C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m)}} \end{aligned}$$

D'après notre hypothèse, $\alpha C_{(m)} \geq \alpha C_{(k)}$:

$$\begin{aligned} c_{\alpha H^\infty} &\leq \frac{(1+\alpha) r_{(k)} + \alpha C_{(m)} + C_{(k)} + (1+\alpha) \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m-1)} + C_{(m)}}{r_{(k)} + \alpha C_{(k)} + C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m)}} \\ c_{\alpha H^\infty} &\leq \frac{(1+\alpha) r_{(k)} + (1+\alpha) C_{(m)} + C_{(k)} + (1+\alpha) \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m-1)}}{r_{(k)} + \alpha C_{(k)} + C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m)}} \end{aligned}$$

En utilisant encore la propriété 5, $\alpha + \alpha^2 = 1$:

$$c_{\alpha H^\infty} \leq \frac{(1+\alpha)(r_{(k)} + C_{(m)} + \sum_{i=k+1}^{m-2} C_{(i)}) + (1+\alpha)\alpha C_{(k)} + (1+\alpha)\alpha C_{(m-1)}}{r_{(k)} + \alpha C_{(k)} + C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m)}}$$

$$c_{\alpha H^\infty} \leq (1+\alpha)\frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + \alpha C_{(m-1)} + C_{(m)}}{r_{(k)} + \alpha C_{(k)} + C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m)}}$$

D'après notre hypothèse nous avons posé aussi que $C_{(k)} \geq \alpha C_{(m-1)}$:

$$c_{\alpha H^\infty} \leq (1+\alpha)\frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + \alpha C_{(m-1)} + C_{(m)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + \alpha C_{(m-1)} + C_{(m)}}$$

$$c_{\alpha H^\infty} \leq 1 + \alpha$$

– $C_{(m-1)} \geq C_{(k)} \geq C_{(m)}$:

Dans la séquence d'ordonnancement construite par l'algorithme optimal (l'adversaire), deux sous-cas sont à considérer : suivant que l'algorithme optimal ordonnance les tâches $\tau_{(m-1)}$ et $\tau_{(m)}$ dans un unique lot ou non.
Supposons dans un premier temps que ces deux tâches sont ordonnancées dans un unique lot. Comme $C_{(m-1)} \geq C_{(m)}$, un lot est exécuté après l'instant $r_{(m)}$ (il faut attendre que les deux tâches soient actives) et pendant une durée de $C_{(m-1)}$.

$$\sigma^*(I) \geq r_{(m)} + C_{(m-1)}$$

Dès lors que le lot B_k est le dernier lot régulier dans la séquence d'ordonnancement construite par l'algorithme en-ligne, (*cf.* figure 12.4), alors :

$$r_{(m)} \geq r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)}$$

$$\sigma^*(I) \geq r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)}$$

En utilisant l'ensemble de ces valeurs, le ratio de compétitivité de ce sous-cas est de :

$$c_{\alpha H^\infty} \leq \frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m} C_{(i)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)}}$$

$$c_{\alpha H^\infty} \leq \frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)} + C_{(m)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)}}$$

$$c_{\alpha H^\infty} \leq 1 + \frac{C_{(m)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)}}$$

En utilisant la propriété 5, $\alpha + \alpha^2 = 1$:

$$\begin{aligned} c_{\alpha H^\infty} &\leq 1 + \alpha \frac{(1+\alpha)C_{(m)}}{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)}} \\ c_{\alpha H^\infty} &\leq 1 + \alpha \frac{(1+\alpha)C_{(m)}}{r_{(k)} + (1+\alpha) C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m-1)}} \end{aligned}$$

D'après le cas dans lequel nous nous sommes placés, $C_{(k)} \geq C_{(m)}$:

$$\begin{aligned} c_{\alpha H^\infty} &\leq 1 + \alpha \frac{(1+\alpha)C_{(k)}}{r_{(k)} + (1+\alpha) C_{(k)} + \sum_{i=k+1}^{m-2} C_{(i)} + C_{(m-1)}} \\ c_{\alpha H^\infty} &\leq 1 + \alpha \end{aligned}$$

Si maintenant les tâches $\tau_{(m-1)}$ et $\tau_{(m)}$ ne sont pas ordonnancées dans un même lot par l'algorithme optimal, par conséquent, après l'arrivée de la tâche $\tau_{(m-1)}$, au moins deux lots sont ordonnancés et le premier lot a pour longueur au moins $C_{(m-1)}$ (car il contient la tâche $\tau_{(m-1)}$) et $C_{(m)}$ (car il contient la tâche $\tau_{(m)}$). Par conséquent :

$$\sigma^*(I) \geq r_{(m-1)} + C_{(m-1)} + C_{(m)}$$

D'après la figure 12.4, la tâche $\tau_{(m)}$ (comme elle n'est pas ordonnancée dans le lot B_{m-1}) arrive dans le système après la fin de l'attente liée à ce lot :

$$r_{(m)} \geq r_{(m-1)} + \alpha C_{(m-1)}$$

En réitérant ce raisonnement jusqu'à la tâche $\tau_{(k+1)}$, nous obtenons :

$$\begin{aligned} r_{(m-1)} &\geq r_{(m-2)} + \alpha C_{(m-2)} \\ &\dots \\ r_{(k+1)} &\geq r_{(k)} + \alpha C_{(k)} \\ r_{(m-1)} &\geq r_{(k)} + \alpha C_{(k)} + ... + \alpha C_{(m-2)} \\ r_{(m-1)} &\geq r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} \end{aligned}$$

Par conséquent, pour le calcul de la longueur d'ordonnancement obtenue avec l'adversaire, nous obtenons :

$$\begin{aligned} \sigma^*(I) &\geq r_{(m-1)} + C_{(m-1)} + C_{(m)} \\ \sigma^*(I) &\geq r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)} \end{aligned}$$

Le ratio de compétitivité déduit est :

$$c_{\alpha H^\infty} \leq \frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m} C_{(i)}}{r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)}}$$

Étant donné que $\alpha r_{(k)} \geq 0$ et $\alpha C_{(m)} \geq 0$, nous pouvons ajouter ces termes au numérateur pour définir le ratio de compétitivité :

$$\begin{aligned}
c_{\alpha H^\infty} &\leq \frac{r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m} C_{(i)} + \alpha r_{(k)} + \alpha C_{(m)}}{r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)}} \\
c_{\alpha H^\infty} &\leq \frac{(1+\alpha) r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m} C_{(i)} + \alpha C_{(m)}}{r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)}} \\
c_{\alpha H^\infty} &\leq \frac{(1+\alpha) r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)} + C_{(m)} + \alpha C_{(m)}}{r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)}} \\
c_{\alpha H^\infty} &\leq \frac{(1+\alpha) r_{(k)} + \alpha C_{(k)} + \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)} + (1+\alpha) C_{(m)}}{r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)}}
\end{aligned}$$

La propriété 5 ($\alpha + \alpha^2 = 1$) nous permet de simplifier le ratio :

$$c_{\alpha H^\infty} \leq \frac{(1+\alpha) r_{(k)} + \alpha C_{(k)} + (1+\alpha)\alpha \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)} + (1+\alpha) C_{(m)}}{r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)}}$$

Or d'après notre cas d'étude, $C_{(m-1)} \geq C_{(k)}$, ce qui nous permet de conclure :

$$\begin{aligned}
c_{\alpha H^\infty} &\leq \frac{(1+\alpha) r_{(k)} + \alpha C_{(m-1)} + (1+\alpha)\alpha \sum_{i=k}^{m-2} C_{(i)} + C_{(m-1)} + (1+\alpha) C_{(m)}}{r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)}} \\
c_{\alpha H^\infty} &\leq \frac{(1+\alpha) r_{(k)} + (1+\alpha) C_{(m-1)} + (1+\alpha)\alpha \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}}{r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)}} \\
c_{\alpha H^\infty} &\leq (1+\alpha) \frac{r_{(k)} + C_{(m-1)} + \alpha \sum_{i=k}^{m-2} C_{(i)} + C_{(m)}}{r_{(k)} + \sum_{i=k}^{m-2} \alpha C_{(i)} + C_{(m-1)} + C_{(m)}} \\
c_{\alpha H^\infty} &\leq 1 + \alpha
\end{aligned}$$

□

Pour conclure, αH^∞ est l'un des meilleurs algorithmes en-ligne pour le problème d'ordonnancement $1|p-batch, r_i, b = \infty|C_{max}$, au même titre que l'algorithme H^∞ mais en insérant moins de temps creux.

12.6. Simulation

Ce paragraphe a pour but de comparer le comportement des trois algorithmes que nous venons de présenter avec l'algorithme H^∞. Ainsi, nous avons généré trente mille configurations de tâches et nous établissons pour chaque configuration quelques valeurs statistiques.

12.6.1. *Présentation du simulateur*

Ce simulateur a été implémenté dans un premier temps, pour chercher des exemples démontrant que les algorithmes αH et $\alpha H2$ ne font pas partie des meilleurs algorithmes en-ligne

pour le problème général. Ensuite, son utilisation a été élargie afin de pouvoir comparer les différents algorithmes en-ligne établis au cours de cette étude avec l'algorithme H^{∞}.

Le simulateur comprend tout d'abord quatre modules, un pour chacun des algorithmes étudiés (H^{∞}, αH, $\alpha H2$ et αH^{∞}). Un autre module est nécessaire car il ne faut pas oublier que pour calculer le ratio de compétitivité de chacun de ces algorithmes sur chaque configuration, nous avons besoin d'une valeur pour l'adversaire, l'algorithme optimal de LEE et UZSOY [LEE 99] (*cf.* paragraphe 11.3).

Pour finir, le module principal crée une configuration contenant de trois à sept tâches. Chaque tâche dispose d'une durée d'exécution aléatoire comprise entre 1 et 12. La première tâche arrive à l'instant 0, puis chacune d'elle arrive après la tâche qui la précède mais avant la date correspondant à la somme entre la date d'arrivée de la tâche qui la précède plus le tiers du temps processeur de cette tâche.

Ensuite, chaque algorithme calcule sa longueur d'ordonnancement puis son ratio de compétitivité. Finalement, trois valeurs statistiques sont calculées : le pourcentage de fois où chaque algorithme est le meilleur, la moyenne et l'écart type du ratio de compétitivité de chaque algorithme en-ligne.

12.6.2. *Résultats de l'expérimentation*

Trois lots de dix mille configurations de tâches ont été lancés. Ces trois lots étaient tous semblables, mais les seules petites nuances permettant de les différencier sont que dans le premier jeu de tests lancé, le simulateur a ressorti une instance selon laquelle, αH a un ratio de compétitivité supérieur à $1 + \alpha$. Dans le second jeu, c'est $\alpha H2$ qui a eu une instance telle que son ratio était supérieur à $1+\alpha$. Enfin, le troisième jeu de tests s'est déroulé normalement. Nous avons récupéré les valeurs (pourcentages, moyennes, écarts types) pour les trois jeux de tests et nous en avons fait la moyenne. Nous avons obtenu les résultats suivants :

Algorithmes	H^{∞}	αH	$\alpha H2$	αH^{∞}
Pourcentages	11.22	29.05	81.82	76.78
Moyennes	1.446316	1.272546	1.186733	1.185980
Écarts types	0.183494	0.097671	0.122591	0.121577

Tableau 12.1. *Comparaisons des algorithmes en-ligne*

La première ligne du tableau 12.1 regroupe les pourcentages où chaque algorithme a été le meilleur. La somme de tous les pourcentages ne donne pas 100% car en effet, plusieurs algorithmes peuvent, lors d'une même configuration, être les meilleurs. Le meilleur en terme de ratio est $\alpha H2$, ce qui n'est pas surprenant puisqu'il essaye toujours d'ordonnancer au plus tôt. Vient ensuite αH^{∞} suivit de αH et H^{∞} qui à contrario de αH et $\alpha H2$ est l'un des meilleurs algorithmes en-ligne pour le problème de la minimisation de la longueur d'exécution sur une machine à traitement par lot où les lots sont de capacité infinie. Cette remarque est

intéressante car malgré ses qualités, il n'empêche qu'H^∞ n'arrive que très rarement à avoir le plus petit ratio de compétitivité (vis-à-vis de l'algorithme optimal hors-ligne).

Ces pourcentages nous permettent de remarquer que l'algorithme H^∞ attend plus longtemps que les autres algorithmes avant de commencer son exécution, ainsi, il génère moins de lots, mais ils sont plus espacés dans le temps. Il obtient ainsi rarement la séquence d'exécution la plus courte. À l'inverse, l'algorithme αH^∞ attend peu mais génère plus de lots, ce qui ne l'empêche pas d'avoir souvent (trois fois sur quatre) la plus courte séquence d'exécution.

La seconde ligne du tableau 12.1 donne les valeurs moyennes des ratios de compétitivité atteintes par nos algorithmes sur l'ensemble des trois jeux de tests. Les algorithmes $\alpha H2$ et αH^∞ sont proches. Comme précédemment, c'est l'algorithme H^∞ qui, même s'il appartient à l'ensemble des meilleurs algorithmes en-ligne pour le problème donné, a le moins bon résultat.

La troisième ligne du tableau 12.1 qui présente les valeurs moyennes des écarts types, montre que les valeurs des ratios de compétitivité de αH restent assez proches. Cet écart grandit un peu pour $\alpha H2$ et αH^∞, avant d'augmenter encore plus pour H^∞.

12.6.3. *Conclusion*

Même si ces résultats sont dépendants de notre environnement de simulation, nous nous rendons compte que même si l'algorithme H^∞ est l'un des meilleurs algorithmes possibles, il n'atteint pas les résultats obtenus par αH et $\alpha H2$. En fait, seul αH^∞, qui est lui aussi l'un des meilleurs algorithmes en-ligne pour le problème de la minimisation de la date maximum de fin d'exécution sur une machine à traitement par lot où les lots sont de capacité infinie, tient la comparaison avec αH et $\alpha H2$.

12.7. Unification

Pendant la relecture de l'article [RID 06], des relecteurs de la revue *EJOR* nous ont indiqué une référence qui a été publiée durant le processus de relecture de notre travail [POO 05]. Dans cet article, les auteurs proposent pour la durée du délai d'attente avant le début de l'exécution d'un lot, un intervalle de temps qui garantit à l'algorithme que quelle que soit la valeur prise dans cet intervalle, il est l'un des meilleurs algorithmes en-ligne pour ce problème. Dans ce paragraphe, nous développons ce résultat.

Remarque 23 *L'algorithme unifié, avant de lancer un lot, choisit une valeur aléatoire dans l'intervalle* $[\alpha C_k, (1+\alpha)r_k + \alpha C_k]$. *On remarque que le maximum correspond au délai d'attente de l'algorithme de* ZHANG, H^∞ *(cf. paragraphe 11.4.2, page 154) et que la limite inférieure est indépendante des dates d'arrivée des tâches. En dehors de ces valeurs, l'algorithme ne peut atteindre la garantie de performance de* $\frac{1+\sqrt{5}}{2}$. *Notons également que cet algorithme peut conduire à un ordonnancement plus court que celui de notre algorithme* αH^∞.

Algorithme 7 : L'algorithme unifié [POO 05]

Donnée : I, une configuration de tâches.
Donnée : $U(0)$, l'ensemble des tâches libre à l'instant 0.
Résultat : Ordonnancement de I par l'algorithme unifié.
Début
  $t \longleftarrow 0$;
  Tant que *Vrai* **faire**
    $k \longleftarrow h$ tel que $\tau_h \in U(t)$ et $C_h = \max\{C_j \mid \tau_j \in U(t)\}$;
    γ une valeur aléatoire prise dans l'intervalle $[\alpha C_k, (1+\alpha)r_k + \alpha C_k]$;
    $s \longleftarrow \max\{t, \gamma\}$;
    Durant *l'intervalle de temps* $[t, s]$ **faire**
      Pour chaque *tâche* τ_h *qui arrive dans le système à l'instant* t' **faire**
        Si $C_h > C_k$ **alors**
          $k \longleftarrow h$;
          γ une valeur prise dans l'intervalle $[\alpha C_k, (1+\alpha)r_k + \alpha C_k]$;
          $t \longleftarrow t'$;
          $s \longleftarrow \max\{t, \gamma\}$;
      $U(t) \longleftarrow U(t) \cup \{\tau_h\}$;
    Ordonnancement dans un même lot de l'ensemble de toutes les tâches disponibles $U(s)$;
    Si *une tâche arrive pendant l'exécution du lot* **alors**
      $t \longleftarrow s + C_k$;
    sinon
      $t \longleftarrow r_h$, où τ_h est la prochaine tâche qui s'active;
fin

Exemple 24 *Ordonnancement de la configuration I (tableau 11.1, page 153) par l'algorithme unifié en supposant que la valeur aléatoire prise est toujours égale à αC_k. La tâche τ_1 arrive dès l'instant 0 et introduit un délai d'attente jusqu'à l'instant $\alpha C_1 = 2\alpha = 1.23$. La tâche τ_2 arrive à l'instant 1. Son temps processeur est égal à celui de τ_1 donc le début d'exécution du premier lot n'est pas plus retardé. Ce premier lot est exécuté à l'instant 1.23, contenant les tâches τ_1 et τ_2 et finit son exécution à l'instant 3.23. La tâche τ_3 activée pendant l'exécution du premier lot, à l'instant 2 et retarde le prochain lot jusqu'à l'instant $\alpha C_3 = 1.85$ qui est passé, donc le lot est exécuté immédiatement après le premier lot, à l'instant 3.23 jusqu'à l'instant 6.23 et contenant juste τ_3. Les tâches τ_4 et τ_5 se sont activées pendant l'exécution du second lot. Le lot ne doit pas être exécuté avant l'instant $\alpha C_4 = 2.47$ déjà passé donc, le troisième lot contenant les tâches τ_4 et τ_5 est exécuté à l'instant 6.23 et est complété à l'instant 10.23. Cet ordonnancement est représenté par la figure 12.5.*

La garantie de performance de l'algorithme unifié est de $\frac{1+\sqrt{5}}{2}$. Cette démonstration est présentée dans l'annexe D.

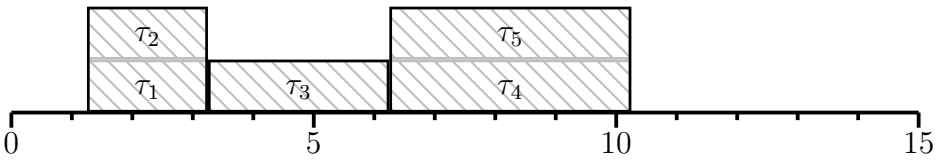

Figure 12.5. *Ordonnancement construit par l'algorithme unifié (avec $\gamma = \alpha C_k$) pour la configuration I, tableau 11.1.*

12.8. Bibliographie

[BAP 00] BAPTISTE P., « Batching Identical Jobs », *Mathematics methods of Operations Research*, vol. 53, p. 355-367, 2000.

[IKU 86] IKURA Y., GIMPLE M., « Efficient scheduling algorithms for a single batch processing machine », *Operations Research Letters*, vol. 5, p. 61-65, 1986.

[LEE 99] LEE C., UZSOY R., « Minimizing makespan on a single batch processing machine with dynamic job arrivals », *International Journal of Production Research*, vol. 37, n°1, p. 219-236, 1999.

[POO 05] POON C., YU W., « A flexible on-line scheduling algorithms for batch machine with infinite capacity », *Annals of Operations Research*, vol. 133, p. 175-181, 2005.

[RID 06] RIDOUARD F., RICHARD P., MARTINEAU P., « On-line scheduling on a batch with unbounded batch size to minimize the makespan », *European Journal of Operational Research (EJOR), to appear*, 2006.

[ZHA 01] ZHANG G., CAI X., WONG C., « On-line algorithms for minimizing makespan on batch processing machines », *Naval Research Logistics*, vol. 48, p. 241-258, 2001.

Chapitre 13

Conclusion sur les machines à traitement par lot

Notre étude a pour objet d'étudier le problème de l'ordonnancement de machines à traitement par lot de capacité infinie et pour la minimisation de la longueur d'ordonnancement. Pour ce problème, il a déjà été démontré que les algorithmes en-ligne ne peuvent pas être meilleurs que 2-compétitifs s'ils n'insèrent pas de temps creux dans leur séquence d'exécution. Nous avons prouvé qu'il est préférable d'insérer des temps creux en montrant que dans ce cas, les algorithmes en-ligne sont alors au mieux $(1+\sqrt{5}/2)$-compétitifs pour l'ordonnancement par une machine à traitement par lot de capacité finie ou infinie, même lorsque les tâches sont toutes de même durée.

Nous avons présenté deux algorithmes en-ligne αH et $\alpha H2$ qui ne font pas partie des meilleurs algorithmes en-ligne pour le problème d'ordonnancement $1|p-batch, r_i, b = \infty|C_{max}$. Mais qui, dans certains cas spécifiques, ont des résultats positifs :

– si toutes les tâches de la configuration ont le même temps processeur, nous avons établi que les algorithmes d'ordonnancement en-ligne αH et $\alpha H2$ font partie des meilleurs algorithmes en-ligne possibles si $\beta = (-1+\sqrt{5})/2$. Nous avons également démontré que n'importe quel algorithme qui décide d'attendre jusqu'à un instant compris entre $\min_{\tau_k \in U(t)}(r_k + \alpha C_k)$ et $\max_{\tau_k \in U(t)}(r_k + \alpha C_k)$ (où $U(t)$ désigne l'ensemble des tâches disponibles à l'instant t) alors il fait également partie des meilleurs algorithmes en-ligne pour le même problème ;

– nous avons également fait apparaître à propos de l'algorithme αH qu'il est l'un des meilleurs algorithmes en-ligne pour le problème d'ordonnancement $1|p-batch, r_i, b = \infty|C_{max}$ lorsque les tâches ont des temps processeurs agréables. Ce qui n'est pas le cas pour l'algorithme $\alpha H2$;

– finalement, pour l'ordonnancement des configurations de tâches n'ayant uniquement que deux dates d'activation, nous avons démontré que l'algorithme d'ordonnancement αH est encore l'un des meilleurs algorithmes en-ligne pour le problème $1|p-batch, b = \infty|C_{max}$. Ce qui n'est encore une fois pas vrai pour l'algorithme $\alpha H2$.

Nous avons, pour le problème d'ordonnancement $1|p-batch, r_i, b = \infty|C_{max}$, établi un algorithme, αH^{∞}, qui fait partie des meilleurs algorithmes en-ligne comme l'algorithme

H^{∞} de [ZHA 01] mais qui a un délai d'attente moins important que celui de l'algorithme H^{∞}. De plus, avec la simulation effectuée, nous pensons que αH^{∞} dans une majorité des cas a un ratio de compétitivité inférieur à celui de H^{∞}.

L'établissement de la borne inférieure et le résultat sur le cas spécifique des configurations de tâches avec seulement deux dates d'activation ont été présentés dans une conférence internationale [RIC 03]. Les autres résultats ont fait l'objet d'une autre conférence internationale [RID 04] ainsi que d'une revue [RID 06].

Récemment, une conclusion a été apportée dans [POO 05a] pour le problème d'ordonnancement en-ligne, $1|p - batch, r_i, b = \infty|C_{max}$. Ces auteurs présentent un panel d'algorithmes en-ligne avec une performance garantie optimale ($(1 + \sqrt{5})/2$-compétitif) incluant les algorithmes αH^{∞} et H^{∞}.

Pour de futurs travaux sur l'ordonnancement par une machine à traitement par lot, il reste à étudier :

– le problème d'ordonnancement par une machine à traitement par lot à capacité finie. Concernant ce problème, un algorithme en-ligne déterministe 2-compétitif est connu. Mais nous ne savons pas s'il existe un meilleur algorithme en-ligne. Dans [ZHA 01] une extension de l'algorithme H^{∞} est proposée mais ces auteurs conjecturent seulement que leur algorithme est $(1 + \sqrt{5})/2$-compétitif. Récemment, dans [POO 05b] un algorithme en-ligne est proposé pour les machines à traitement par lot et à capacité finie avec une performance garantie de 2 et qui est aussi $7/4$-compétitif dans le cas de machine à capacité 2.

– d'autres critères d'optimisation : la minimisation du retard maximum ou le nombre de tâches en retard.

13.1. Bibliographie

[POO 05a] POON C., YU W., « A flexible on-line scheduling algorithms for batch machine with infinite capacity », *Annals of Operations Research*, vol. 133, p. 175-181, 2005.

[POO 05b] POON C., YU W., « On-line scheduling algorithms for a batch machine with finite capacity », *Journal of Combinatorial Optimization*, vol. 9, n°2, p. 167-186, 2005.

[RIC 03] RICHARD P., RIDOUARD F., MARTINEAU P., « On-line scheduling on a single batching machine to minimize the makespan », 6^{th} *Int. Conference on Industrial Engineering and Production Management (IEPM'03), Porto (Portugal)*, , n°26-28, may 2003.

[RID 04] RIDOUARD F., RICHARD P., MARTINEAU P., « On-line minimization of makespan for single batching machine scheduling problems », 9^{th} *International Workshop on Project Management and Scheduling (PMS'04)*, p. 287-290, 2004.

[RID 06] RIDOUARD F., RICHARD P., MARTINEAU P., « On-line scheduling on a batch with unbounded batch size to minimize the makespan », *European Journal of Operational Research (EJOR), to appear*, 2006.

[ZHA 01] ZHANG G., CAI X., WONG C., « On-line algorithms for minimizing makespan on batch processing machines », *Naval Research Logistics*, vol. 48, p. 241-258, 2001.

Conclusion

TROISIÈME PARTIE

Conclusion générale

Chapitre 14

Conclusion générale

Lors de cette étude, nous avons cherché des solutions à deux problèmes d'ordonnancement :

– l'ordonnancement de tâches à suspension ;

– l'ordonnancement de tâches par une machine à traitement par lot.

L'ordonnancement de tâches à suspension se fait dans un contexte temps réel. Tandis que l'ordonnancement par une machine à traitement se fait dans un contexte classique d'ordonnancement. Ces deux problèmes ont été étudiés à travers l'analyse de compétitivité des algorithmes en-ligne.

Dans la partie I, nous avons étudié l'ordonnancement de configurations de tâches à suspension. Dans le chapitre 4, nous avons rappelé quelques fondements de l'ordonnancement temps réel monoprocesseur. Nous avons présenté, au chapitre 5, un état de l'art de l'ordonnancement de tâches à suspension. Pour cette étude, nous nous sommes limités aux tâches qui ne se suspendent qu'au plus une seule fois durant leur exécution. Trois principaux axes d'étude ont été abordés au cours de cette partie :

– le chapitre 6 établit la $\mathcal{NP}$-difficulté de l'ordonnancement des tâches à suspension sur un système monoprocesseur ;

– le chapitre 7 étudie la compétitivité des principaux algorithmes en-ligne pour deux critères de performance : la minimisation du nombre de tâches en retard et la minimisation du temps de réponse maximum ;

– finalement, le chapitre 8 établit des bornes du pessimisme engendré par des tests d'ordonnançabilité.

Ainsi, nous avons démontré que le problème d'ordonnancement des tâches à suspension et à échéance sur requête est $\mathcal{NP}$-Difficile au sens fort. Nous avons également montré qu'aucun algorithme en-ligne ne peut être *universel* pour ce problème, sauf si $\mathcal{P} = \mathcal{NP}$. De plus, la présence possible d'anomalies d'ordonnancement a été démontrée pour les algorithmes en-ligne classiques RM, DM, EDF (*cf.* paragraphe 4.2). Finalement, nous avons établi qu'aucun algorithme en-ligne déterministe n'est optimal pour l'ordonnancement de tâches à suspension sporadiques dans un système monoprocesseur.

Puis, dans le chapitre 7, nous avons évalué les algorithmes en-ligne (RM, DM, EDF et LLF (paragraphe 4.2)), en utilisant l'analyse de compétitivité. Ainsi, nous avons démontré qu'ils sont non compétitifs pour minimiser le nombre de tâches en retard et ce, même pour des configurations de tâches à suspension avec des charges processeurs faibles (proches de zéro) alors qu'il existe des ordonnancements faisables de ces configurations. De plus, la technique d'augmentation des ressources n'améliore pas les performances d'EDF. Enfin, en ce qui concerne la minimisation du temps de réponse maximum, nous avons démontré que les algorithmes en-ligne classiques sont au mieux 2-compétitifs.

Dans le dernier chapitre de cette partie, au chapitre 8, nous avons étudié trois tests d'ordonnançabilité, basés sur l'algorithme à priorité fixe RM et pour la minimisation du temps de réponse maximum : les méthodes A et B de KIM [KIM 95] et la méthode de LIU [LIU 00]. Notre étude a démontré qu'il n'existait aucune équivalence entre la méthode A de KIM et le test de LIU. Ainsi, un test peut établir qu'une tâche est ordonnançable et l'autre non, et inversement. En utilisant l'analyse de compétitivité, la borne inférieure du ratio de compétitivité de chaque test est de 2.91667 pour la méthode A de KIM, 2.75 pour la méthode B et 2.875 pour la méthode de LIU. Finalement, une simulation permet de confirmer la meilleure méthode qui est la méthode B de KIM. Mais ces résultats sont à relativiser car notre modèle stochastique utilisé pour la génération des configurations de tâches a été développé pour générer des bornes inférieures du ratio de compétitivité (avec un nombre faible de tâches).

Puisque nous n'avons toujours pas trouvé de solution à ce problème d'ordonnancement, il serait donc intéressant de rechercher un algorithme hors-ligne optimal. Un autre aspect intéressant à ce problème serait de mesurer l'impact des anomalies d'ordonnancement sur le nombre de tâches ne respectant pas ses contraintes temporelles, en utilisant le principe présenté concernant les tâches non préemptibles et sans suspension [MOK 05]. D'autres pistes d'études sont : les deux problèmes déduis de celui-ci : le problème de gigue sur activation et le problème de la suspension à la fin du traitement (p. ex. la latence associée à une opération sur un actionneur), l'étude du problème des tâches à suspension aux tâches dépendantes ainsi qu'aux systèmes multiprocesseurs.

Dans la partie II, nous avons abordé le problème de l'ordonnancement des tâches par une machine à traitement par lot. Dans le chapitre 11, nous avons présenté la bibliographie de ce problème d'ordonnancement. Au chapitre suivant, le chapitre 12, nous avons montré que les algorithmes en-ligne qui insèrent des temps creux dans leur séquence d'ordonnancement sont au mieux $(1+\sqrt{5}/2)$-compétitifs pour l'ordonnancement par une machine à traitement par lot de capacité finie ou infinie et des durées de tâches identiques alors qu'ils ne sont pas mieux que 2-compétitifs dans le cas inverse. De plus, nous avons proposé plusieurs algorithmes en-ligne :

– l'algorithme αH qui ne fait pas partie des meilleurs algorithmes en-ligne pour le problème d'ordonnancement $1|p-batch, r_i, b=\infty|C_{max}$. Mais qui l'est si tous les temps processeurs sont égaux, ou si les temps processeurs sont agréables ou finalement s'il n'y a que deux dates d'activation différentes pour toute la configuration.

– l'algorithme $\alpha H2$, qui attend moins que l'algorithme αH mais qui ne fait pas non plus partie des meilleurs algorithmes en-ligne pour le problème d'ordonnancement $1|p-batch, r_i, b=\infty|C_{max}$ et qui ne l'est pas également si les temps processeurs sont agréables

ou s'il n'y a que deux dates d'activation par configuration, mais qui l'est si tous les temps processeurs sont égaux. De plus, si un algorithme a un temps d'attente avant le début du lancement d'un lot compris entre $\min_{\tau_k \in U(t)}(r_k + \alpha C_k)$ et $\max_{\tau_k \in U(t)}(r_k + \alpha C_k)$ (où $U(t)$ désigne l'ensemble des tâches disponibles à l'instant t) alors il fait également partie des meilleurs algorithmes en-ligne si tous les temps processeurs sont égaux.

– l'algorithme αH^{∞}, qui a une garantie de performance égale à $(1 + \sqrt{5})/2$ pour le problème, $1|p - batch, r_i, b = \infty|C_{max}$, il fait donc partie des meilleurs algorithmes en-ligne comme l'algorithme H^{∞} de [ZHA 01]. Mais le délai d'attente d'αH^{∞} est moins important que celui de l'algorithme H^{∞}. De plus, avec la simulation effectuée, nous pensons que αH^{∞} a, la plupart du temps, un ratio de compétitivité inférieur à celui de H^{∞}. Notons que le problème a été récemment fermé par l'article [POO 05].

Pour de futurs travaux, il reste à étudier : le problème d'ordonnancement par une machine à traitement par lot à capacité finie. Ce problème est toujours ouvert à notre connaissance et où seulement un algorithme en-ligne déterministe 2-compétitif est connu. Mais comme la borne du ratio de compétitivité pour ce problème est de $(1 + \sqrt{5})/2$, il existe probablement un meilleur algorithme en-ligne. Cette étude pourrait être étendue à d'autres critères de performance.

14.1. Bibliographie

[KIM 95] KIM I., CHOI K., PARK S., KIM D., HONG M., « Real-Time Scheduling of tasks that contain the external blocking intervals », *proc. Conference on Real-Time Computing Systems and Applications*, p. 54-59, 1995.

[LIU 00] LIU J., *Real-Time Systems*, Prentice hall, 2000.

[MOK 05] MOK A., POON W.-C., « Non-Preemptive Robustness under Reduced System Load », 26^{th} *IEEE International Real-Time Systems Symposium (RTSS'05)*, vol. 1, n°5-8, p. 200-209, December 2005.

[POO 05] POON C., YU W., « A flexible on-line scheduling algorithms for batch machine with infinite capacity », *Annals of Operations Research*, vol. 133, p. 175-181, 2005.

[ZHA 01] ZHANG G., CAI X., WONG C., « On-line algorithms for minimizing makespan on batch processing machines », *Naval Research Logistics*, vol. 48, p. 241-258, 2001.

ANNEXES

Annexe A

La méthode de PALENCIA pour les algorithmes à priorité fixe

A.1. Introduction

Les études sur l'ordonnancement de tâches avec des algorithmes à priorité fixe a abouti à déterminer des calculs du temps de réponse pour des tâches s'exécutant sur un unique processeur en incluant les effets de synchronisation, avec $D_i \leq T_i$ ou $D_i \geq T_i$, en prenant en compte les contraintes de précédence, les variations de priorité des tâches et les systèmes surchargés.

Mais aucune étude n'a encore trouvé de solution exacte pour le calcul du temps de réponse dans les systèmes distribués et dans les systèmes où les tâches peuvent se suspendre. Quelques travaux ont été menés sur l'ordonnancement à priorité fixe dans les systèmes distribués ou dans les systèmes de tâches à suspension mais les résultats obtenus sur le calcul du pire temps de réponse sont restés très pessimistes ce qui ne permet pas leur utilisation. Pour diminuer ce pessimisme, TINDELL [TIN 94] développa une méthode de calcul du pire temps de réponse introduisant la notion d'*offsets* statiques. Cette méthode est réutilisée par PALENCIA *et al.* dans leurs travaux.

Avant d'évoquer les travaux de PALENCIA et GONZALES-HARBOUR [PAL 98], commençons par décrire le modèle utilisé.

A.1.1. *Modèle de tâche : transaction*

Le système de tâches utilisé par PALENCIA est un système de tâches préemptives et à priorité fixe. Les tâches sont placées dans des *transactions* [1].

Définition 26 *Une transaction est une collection de tâches ayant des caractéristiques communes. Chaque transaction (notée Γ_i) a une période notée T_i et contient m_i tâches. Chaque tâche hérite de sa transaction, sa période. Chaque transaction est réveillée par un événement*

1. *cf.* Rate Monotonic Analysis [KLE 93]

extérieur donc toutes les tâches composant la transaction arrivent au même instant. Mais chaque tâche τ_i à son activation, a un délai d'attente supplémentaire correspondant à la gigue sur activation.

Un autre délai d'attente est rajouté dans le modèle à l'activation de la tâche. Ce temps d'attente appelé offset représente un type de contrainte de précédence entre les tâches, il indique le décalage de réveil de chaque tâche par rapport à l'arrivée de la transaction dans le système.

Définition 27 *Un offset est une durée d'attente entre l'arrivée de la tâche dans le système et son réveil. Il existe deux types d'offsets :*

– offsets statiques : à chaque réveil d'une transaction, les tâches de la transaction gardent les mêmes offsets.

– offsets dynamiques : l'offset des tâches peut varier du réveil d'une transaction à une autre.

Chaque tâche se note $\tau_{i,j}$ pour signifier qu'elle est la j^{ieme} tâche de la transaction Γ_i à laquelle elle appartient.

– Son *offset* est noté $\Phi_{i,j}$;

– sa gigue maximum au démarrage est notée $J_{i,j}$;

– son pire temps processeur requis est noté $C_{i,j}$

– sa limite de fin d'exécution relative à sa date de réveil : $D_{i,j}$;

– son pire temps de réponse se note $R_{i,j}$.

Les tâches peuvent utiliser des sémaphores avec le protocole à priorité plafond. Dans cette hypothèse, les effets des tâches moins prioritaires sur $\tau_{i,j}$ peuvent se limiter à un facteur de blocage noté $B_{i,j}$.

TINDELL [TIN 94] mena son étude avec des offsets statiques et tels que chaque offset soit inférieur à la période de la tâche à laquelle il est associé. Mais pour l'étude des tâches à suspension, les offsets statiques ne nous suffisent pas. En effet, nous verrons que comme la durée de suspension n'est pas constante, les offsets statiques ne sont pas directement utilisables pour représenter les suspensions. De plus, rien dans l'étude des tâches à suspension ne contraint une tâche à se suspendre pour une durée inférieure à celle de la période. Par conséquent, les travaux de TINDELL ne permettent pas directement d'étudier les tâches à suspension. C'est pour cette raison que nous ne détaillons pas plus ces travaux mais nous présentons directement l'extension de ces études apportée par PALENCIA et GONZALEZ HARBOUR qui peut s'appliquer aux traitements des tâches avec suspension.

A.2. Les travaux de PALENCIA et GONZALEZ HARBOUR [PAL 98] sur les offsets statiques

Avant d'étudier les offsets dynamiques et plus particulièrement les tâches à suspension, PALENCIA et GONZALEZ HARBOUR étudièrent dans un premier temps les offsets statiques. À

noter également que PALENCIA et GONZALEZ HARBOUR levèrent la contrainte de restriction sur la longueur des offsets et des gigues sur activation que TINDELL limita à la période de la tâche.

Pour déterminer leur test d'ordonnançabilité, la méthode calcule dans un premier temps, le temps de réponse exact des tâches (à offset statique). Mais la complexité est élevée et le rend inutilisable dès lors que le nombre de tâches du système devient important. Ils en déduisent donc un test approché et une borne supérieure moins coûteuse en terme d'opérations rendant ainsi le test utilisable.

A.2.1. *Calcul exact du temps de réponse*

Le temps de réponse de chaque tâche doit être calculé indépendamment des autres. Soit $\tau_{a,b}$, la tâche dont le pire temps de réponse exact va être calculé. Pour déterminer le pire temps de réponse, nous devons étudier la période d'activité de la tâche $\tau_{a,b}$.

Définition 28 *Une période d'activité de $\tau_{a,b}$ est un intervalle de temps pendant lequel le processeur reste constamment occupé à exécuter $\tau_{a,b}$ et des tâches à plus haute priorité.*

Pour déterminer le pire temps de réponse de la tâche $\tau_{a,b}$, nous devons construire un instant critique t_c.

Définition 29 *Un instant critique définit l'instant de début d'une période d'activité.*

L'instant critique t_c à construire doit correspondre au début de la pire (plus longue) période d'activité de $\tau_{a,b}$. A noter que pour simplifier les calculs et sans perte de généralité, nous posons $t_c = 0$. Les tâches de chaque transaction ne sont pas toutes indépendantes. Ces relations sont représentées par les offsets. A cause des offsets, à l'instant critique, toutes les tâches plus prioritaires que $\tau_{a,b}$ ne peuvent pas simultanément s'activer puisque les arrivées des tâches sont décalées. La construction de cet instant critique signifie qu'à l'instant t_c une tâche de chaque transaction et de priorité supérieure ou égale à celle de $\tau_{a,b}$ s'active.

Pour déterminer le pire temps de réponse de la tâche $\tau_{a,b}$, il faut connaître la pire contribution de chaque transaction Γ_i dans le calcul du pire temps de réponse de la tâche $\tau_{a,b}$.

PALENCIA et GONZALEZ HARBOUR ont démontré que la pire contribution d'une transaction Γ_i survient quand une tâche de la transaction s'active après son délai d'attente correspondant à la gigue d'activation à l'instant critique t_c. Ainsi, pour chaque transaction, notons $\tau_{i,k}$, la tâche dont la première apparition dans la période d'activité de $\tau_{a,b}$ coïncide avec l'instant critique (après avoir subi un délai d'attente correspondant à sa gigue et son offset maximum). À noter que la priorité de $\tau_{i,k}$ est supérieure à celle de $\tau_{a,b}$ car pendant une période d'activité de $\tau_{a,b}$, seule la tâche $\tau_{a,b}$ et les tâches plus prioritaires peuvent s'exécuter.

La variable $\varphi_{i,j,k}$ définit la différence de temps entre la première date d'activation d'une requête dans la période d'activité et l'instant critique généré par la tâche $\tau_{i,k}$. Sa valeur est :

$$\varphi_{i,j,k} = T_i - ((\Phi_{i,k}\ mod\ T_i) + J_{i,k} - (\Phi_{i,j}\ mod\ T_i))\ mod\ T_i$$

La contribution de la transaction Γ_i (dont la tâche $\tau_{i,k}$ s'active à l'instant t_c) dans le calcul du pire temps de réponse $\tau_{a,b}$ pour une période d'activité de longueur t, est donnée par la formule suivante :

$$W_{i,k}(\tau_{a,b}, t) = \sum_{\forall j \in hp_i(\tau_{a,b})} \left(\left\lfloor \frac{J_{i,j} + \varphi_{i,j,k}}{T_i} \right\rfloor + \left\lceil \frac{t - \varphi_{i,j,k}}{T_i} \right\rceil \right) C_{i,j} \quad \text{(A.1)}$$

et $hp_i(\tau_{a,b})$ étant l'ensemble des tâches de Γ_i qui ont une priorité plus élevée que celle de $\tau_{a,b}$.

Pour obtenir le pire temps de réponse, il faut pour chaque transaction Γ_i du système, rechercher la tâche $\tau_{i,k}$ qui, arrivant dans le système à l'instant critique, maximisera la contribution de la transaction Γ_i dans le pire temps de réponse de $\tau_{a,b}$ (*cf.* formule A.1). Notons $\nu(i)$, l'indice k de la tâche de la transaction Γ_i arrivant à l'instant critique.

Définition 30 *Notons par $\nu = (\nu(1), \nu(2), \dots)$, une combinaison possible telle que pour chaque transaction Γ_i, la tâche $\tau_{i,\nu(i)}$ s'active à critique t_c.*

Pour obtenir le pire temps de réponse exact de chaque tâche, il faut essayer toutes les combinaisons possibles ν et conserver la combinaison menant au pire temps de réponse. Par conséquent, si nous notons $N_i(\tau_{a,b})$, le cardinal de $hp_i(\tau_{a,b})$, et si nous n'oublions pas de vérifier le cas où $\tau_{a,b}$ arrive elle-même à l'instant critique t_c alors l'ensemble de toutes les combinaisons possibles (ou scénarios possibles) est :

$$\begin{aligned} N_\nu(\tau_{a,b}) &= (N_a(\tau_{a,b}) + 1)\ N_1(\tau_{a,b})\ N_2(\tau_{a,b}) \dots \\ &= (N_a(\tau_{a,b}) + 1) \prod_{\forall i \neq a} N_i(\tau_{a,b}) \end{aligned}$$

Plusieurs requêtes de la tâche $\tau_{a,b}$ peuvent se réveiller et s'exécuter pendant la période d'activité. Ainsi, nous numérotons les requêtes de la tâche $\tau_{a,b}$ qui s'exécutent pendant la période d'activité, en utilisant la lettre p et par ordre croissant de leur date d'activation. La première activation après l'instant critique (dans l'intervalle $[0, T_a]$) est notée $p = 1$. Et les autres activations sont numérotées en conséquence. C'est-à-dire que l'activation ayant lieu dans l'intervalle $(T_i, 2T_i]$ est $p = 2$ ainsi de suite par celles qui suivent. Dans l'intervalle $(-T_i, 0]$, c'est l'apparition $p = -1$ et de même pour les précédentes.

Pour chaque combinaison ν et chaque requête p de la tâche $\tau_{a,b}$ susceptible d'être présente dans la période d'activité, nous devons calculer sa date de fin d'exécution :

$$w_{a,b}^{\nu}(p) = B_{a,b} + (p - p_{0,a,b}^{\nu} + 1)C_{a,b} + \sum_{\forall i} W_{i,\nu(i)}\left(\tau_{a,b}, w_{a,b}^{\nu}(p)\right)$$

où $p_{0,a,b}^{\nu}$ désigne le plus petit indice p de la requête de $\tau_{a,b}$ qui est susceptible de s'exécuter pendant la période d'activité et qui vaut :

$$p_{0,a,b}^{\nu} = -\left\lfloor \frac{J_{a,b} + \varphi_{a,b,\nu(a)}}{T_a} \right\rfloor + 1$$

La longueur de la période d'activité est :

$$L_{a,b}^{\nu} = B_{a,b} + \left(\left\lceil \frac{L_{a,b}^{\nu} - \varphi_{a,b,\nu(a)}}{T_a} \right\rceil - p_{0,a,b}^{\nu} + 1 \right) C_{a,b} + \sum_{\forall i} W_{i,\nu(i)}\left(\tau_{a,b}, L_{a,b}^{\nu}\right)$$

En conséquence, le plus grand indice p des requêtes de $\tau_{a,b}$ qui doit être pris en compte s'en déduit :

$$P_{L,a,b}^{\nu} = \left\lceil \frac{L_{a,b}^{\nu}(p) - \varphi_{a,b,\nu(a)}}{T_a} \right\rceil$$

Le temps de réponse de la requête p de $\tau_{a,b}$ est :

$$R_{a,b}^{\nu}(p) = w_{a,b}^{\nu}(p) - \varphi_{a,b,\nu(a)} - (p-1)T_a + \Phi_{a,b}$$

Et pour avoir le pire temps de réponse, il faut prendre la valeur maximum sur l'ensemble des requêtes p et sur l'ensemble des combinaisons ce qui donne la formule finale du pire temps de réponse de la tâche $\tau_{a,b}$:

$$R_{a,b} = \max_{\forall \nu} \left[\max_{p=P_{0,a,b}^{\nu} \cdots P_{L,a,b}^{\nu}} \left(R_{a,b}^{\nu}(p)\right) \right]$$

Pour finir, à chaque tâche du système il faut appliquer cette formule et vérifier que la valeur obtenue est inférieure à l'échéance relative. Cette méthode, avec le nombre de calculs qu'elle suscite, est exponentielle et est par conséquent peu utilisable. La solution est donc de calculer une borne supérieure la plus précise possible mais en limitant le nombre de calculs.

A.2.2. *Une borne supérieure du temps de réponse*

Afin d'en diminuer la complexité, PALENCIA et GONZALEZ HARBOUR ont commencé par diminuer la complexité des calculs des contributions de chaque transaction dans le calcul du pire temps de réponse, en rendant le calcul de chaque transaction indépendant des autres transactions. Ce calcul ne sera plus exact mais deviendra une borne supérieure des interférences dues à une transaction Γ_i sur le temps de réponse de la tâche $\tau_{a,b}$.

Au lieu de parcourir l'ensemble des combinaisons possibles, une solution est de prendre pour chaque transaction la tâche $\tau_{i,k}$ qui donnera la plus importante interférence ou en terme de calculs, la plus grande valeur $W_{i,k}$. Ce qui donne pour borne supérieure des interférences :

$$W_i^*(\tau_{a,b}, t) = \max_{\forall \tau_{i,k} \in hp_i(\tau_{a,b})} W_{i,k}(\tau_{a,b}, t)$$

où t désigne la longueur de la période d'activité.

Pour éviter d'introduire trop de pessimisme, cette solution est utilisée pour toutes les transactions sauf celle contenant la tâche dont le pire temps de réponse doit être calculé.

Le pire temps de réponse de la requête p de $\tau_{a,b}$ si l'instant critique est introduit pour la transaction Γ_a par la tâche $\tau_{a,c}$ est :

$$\begin{aligned} w_{a,b,c}(p) = B_{a,b} &+ (p - p_{0,a,b,c} + 1)C_{a,b} \\ &+ W_{a,c}\left(\tau_{a,b}, w_{a,b,c}(p)\right) \\ &+ \sum_{\forall i \neq a} W_i^*\left(\tau_{a,b}, w_{a,b,c}(p)\right) \end{aligned}$$

Le paramètre $p_{0,a,b,c}$ qui correspond à l'index de la première requête de la tâche $\tau_{a,b}$ qui s'active pendant la période d'activité est :

$$p_{0,a,b,c} = -\left\lfloor \frac{J_{a,b} + \varphi_{a,b,c}}{T_a} \right\rfloor + 1$$

La borne supérieure de la longueur de la période d'activité est :

$$\begin{aligned} L_{a,b,c} = B_{a,b} &+ \left(\left\lceil \frac{L_{a,b,c} - \varphi_{a,b,c}}{T_a} \right\rceil - p_{0,a,b,c} + 1 \right) C_{a,b} \\ &+ W_{a,c}\left(\tau_{a,b}, w_{a,b,c}(p)\right) \\ &+ \sum_{\forall i \neq a} W_i^*\left(\tau_{a,b}, L_{a,b,c}\right) \end{aligned}$$

et le plus grand indice des requêtes de $\tau_{a,b}$ qui s'active dans la période d'activité :

$$p_{L,a,b,c} = \left\lceil \frac{L_{a,b,c} - \varphi_{a,b,c}}{T_a} \right\rceil$$

La borne supérieure du temps de réponse de la requête p de $\tau_{a,b}$ est :

$$R_{a,b,c}(p) = w_{a,b,c}(p) - \varphi_{a,b,c)} - (p-1)T_a + \Phi_{a,b}$$

Et ainsi, la borne supérieure du pire temps de réponse d'une tâche $\tau_{a,b}$ est :

$$R_{a,b} = \max_{\forall c \in hp_a(\tau_{a,b}) \cup b} \left[\max_{p=P_{0,a,b,c}..P_{L,a,b,c}} (R_{a,b,c}(p)) \right] \tag{A.2}$$

Enfin, pour savoir si une configuration de tâches est ordonnançable, il faut calculer pour toutes les tâches $\tau_{a,b}$ la borne supérieure du pire temps de réponse $R_{a,b}$ et vérifier que cette borne est bien inférieure à son échéance relative à sa date de réveil, $D_{a,b}$.

A.3. Les tâches à suspension et offsets dynamiques

Les offsets dynamiques sont utilisés pour la gestion des systèmes comprenant des tâches à suspension et pour les systèmes distribués. L'idée pour gérer les tâches à suspension est de décomposer ces tâches en plusieurs petites tâches indépendantes. En fait, le processus est le même que celui de KIM *et al.* présenté dans le paragraphe 5.2.4. Pour chaque tâche à suspension τ_i, deux tâches $\tau_{i,1}$ et $\tau_{i,2}$ (une par bloc d'exécution) sont générées et regroupées en une transaction : elles héritent de la tâche initiale la date d'activation et la période. En plus de ces caractéristiques, à chaque tâche est ajouté un offset (pour permettre le décalage d'exécution entre les deux tâches et ainsi conserver les propriétés des tâches à suspension) :

– à la tâche $\tau_{i,1}$, un offset nul car il n'y a pas de relation de précédence. Ainsi $\Phi_{i,1} = 0$;

– pour déterminer l'offset de la tâche $\tau_{i,2}$, deux paramètres sont à considérer : le temps de réponse de la tâche $\tau_{i,1}$ qui varie entre son meilleur temps de réponse, noté $R^b_{i,1}$ et son pire temps de réponse noté $R_{i,1}$ et la durée de suspension de τ_i. PALENCIA et GONZALEZ HARBOUR ne considèrent pas seulement une borne supérieure de la suspension mais également une borne inférieure. Cette borne inférieure se note $X_{i,min}$. Ainsi, l'offset de la tâche générée $\tau_{i,2}$ est comprise dans l'intervalle :

$$\Phi_{i,2} \in [R^b_{i,1} + X_{i,min}, R_{i,1} + X_i] \tag{A.3}$$

Posons $\Phi_{i,2,min} = R^b_{i,1} + X_{i,min}$ et $\Phi_{i,2,max} = R_{i,1} + X_i$.

Ainsi, le problème d'ordonnançabilité de tâches à suspension revient à transformer les tâches à suspension en tâches indépendantes, avec gigue sur activation et soumises à des offsets dynamiques. Mais pour résoudre ce problème, PALENCIA et GONZALEZ HARBOUR transforment les tâches à offset dynamique en tâches avec offset statique. En effet, l'offset des tâches $\tau_{i,1}$ est nul donc il est statique. De plus, à chaque tâche $\tau_{i,2}$ est attribué un offset statique égal à la valeur minimum de l'offset dynamique et il ajoute au délai de gigue d'activation, le temps supplémentaire d'attente subi par la tâche $\tau_{i,2}$ pour son offset. Par conséquent, chaque tâche $\tau_{i,2}$ est transformée en une tâche $\tau_{i,2}^{'}$ à offset statique :

$$\Phi_{i,2}^{'} = \Phi_{i,2,min} = R_{i,1}^{b} + X_{i,min}$$

$$J_{i,2}^{'} = J_i + \Phi_{i,2,max} - \Phi_{i,2,min} = J_i + R_{i,1} - R_{i,1}^{b} + X_i - X_{i,min}$$

Pour la valeur de $R_{i,1}^{b}$, des méthodes de calcul du meilleur temps de réponse existent [GUT 98].

Cependant, dans les systèmes à offset dynamique, les valeurs de $\Phi_{i,2,min}$ et $\Phi_{i,2,max}$ dépendent du temps de réponse de la tâche qui la précède $\tau_{i,1}$, qui dépend à son tour de la valeur des offsets. Ce problème est similaire à celui des systèmes distribués, où la valeur des gigues sur activation dépend des temps de réponse des tâches qui elles-mêmes dépendent des valeurs des gigues sur activation. La solution présentée dans [TIN 94], consiste à prendre au départ des temps de réponse nuls et ensuite à itérer en utilisant les formules A.2 et A.3. L'itération s'arrête au rang n auquel $R_{i,j}^{(n+1)} = R_{i,j}^{(n)}$.

Annexe B

La méthode de PALENCIA pour EDF, basé sur le calcul du pire temps de réponse [PAL 03]

La méthode d'analyse utilisée par PALENCIA et GONZALEZ HARBOUR pour déterminer, si une configuration de tâches est ordonnançable sous EDF, est semblable à celle qu'ils ont utilisée dans [PAL 98] et qui a été présentée dans l'annexe A. En effet, pour calculer le pire temps de réponse d'une tâche, cette méthode analyse sa plus longue période d'activité. La plus longue période d'activité démarre par un instant critique t_c et pour faciliter les calculs, nous supposons $t_c = 0$.

Le modèle de tâche (c.-à-d. transaction de tâches) est le même que celui présenté dans le paragraphe A.1.1.

Soit $\tau_{a,b}$, la tâche dont nous calculons le pire temps de réponse. Cette étude s'intéresse en premier aux offsets statiques puis aux offsets dynamiques. Enfin, l'extension au cas de tâches à suspension sera finalement décrite.

B.1. Offsets statiques

La première étape consiste à évaluer la contribution de chaque transaction Γ_i dans le calcul du pire temps de réponse de $\tau_{a,b}$. Nous savons que cette contribution est au maximum quand la première activation d'une tâche $\tau_{i,k}$ de Γ_i à l'intérieur de la période d'activité coïncide avec le début de celle-ci, après avoir subi le délai maximum de gigue $J_{i,k}$. Pour la suite, chaque transaction Γ_i possède sa tâche $\tau_{i,k}$ avec les propriétés précédentes.

B.1.1. *Calcul exact du temps de réponse*

$\varphi_{i,j,k}$ représente la durée entre l'instant critique, instant auquel $\tau_{i,k}$ fait sa première apparition dans la période d'activité et la première apparition d'une requête de $\tau_{i,j}$ dans la période d'activité :

$$\varphi_{i,j,k} = T_i - (\Phi_{i,k} + J_{i,k} - \Phi_{i,j}) mod T_i$$

La pire contribution d'une tâche $\tau_{i,j}$ de la transaction Γ_i, au pire temps de réponse de la tâche $\tau_{a,b}$, dépend également de l'échéance relative de la tâche $\tau_{a,b}$ car l'algorithme étudié est EDF qui prend ses décisions d'ordonnancement en fonction de l'échéance des tâches. Ainsi, la pire contribution d'une tâche $\tau_{i,j}$ de la transaction Γ_i quand l'activation de $\tau_{i,k}$, dans une période d'activité de longueur t, coïncide avec le début de la période d'activité est :

$$W_{i,j,k}(t, D_{a,b}) = \left(\left\lfloor \frac{J_{i,j}+\varphi_{i,j,k}}{T_i} \right\rfloor + \min \left(\left\lceil \frac{t-\varphi_{i,j,k}}{T_i} \right\rceil , \left\lfloor \frac{D_{a,b}-\varphi_{i,j,k}-D_{i,j}}{T_i} \right\rfloor + 1 \right) \right)_0 C_{i,j}$$

où $(x)_0$ signifie que si x est négatif alors la valeur est 0.

La contribution d'une transaction Γ_i dans une période d'activité de longueur t est la somme des contributions des tâches qui la composent :

$$W_{i,k}(t, D_{a,b}) = \sum_{\tau_{i,j} \in \Gamma_i} W_{i,j,k}(t, D_{a,b})$$

Pour calculer le pire temps de réponse de chaque tâche $\tau_{a,b}$, nous devons (comme dans le paragraphe A.2) calculer la contribution de chacune des transactions. De plus, le choix de la tâche qui pour chaque transaction, commence la période d'activité, influence le temps de réponse. Ainsi, l'ensemble de toutes les combinaisons possibles sur le choix des tâches s'activant à l'instant critique pour chaque transaction doit être exploré. Mais ce nombre de combinaisons possibles est très important, trop important pour que ce test soit utilisable. Ainsi comme précédemment, un test approché, moins coûteux en calculs est mis en place.

B.1.2. *Approximation de la limite supérieure du temps de réponse*

Comme dans la technique d'analyse abordée au paragraphe A.2.2, nous utiliserons une méthode d'approximation qui nous permettra d'obtenir une limite supérieure avec un minimum d'opérations et de scénarios à vérifier.

Pour faire un calcul exact des contributions des différentes transactions, nous devons trouver chaque tâche $\tau_{i,k}$ de chaque transaction Γ_i dont la première activation dans la période d'activité correspond à l'instant t_c et qui mènera au pire temps de réponse. Mais ces calculs sont nombreux donc trop coûteux. Pour simplifier ces calculs, nous prendrons comme borne supérieure, le maximum des interférences causées par chaque transaction Γ_i (indépendamment des autres transactions) en considérant tour à tour chaque tâche de la transaction comme celle s'activant à l'instant critique :

$$W_i^* = \max \left(W_{i,k}(t, D_{a,b}) \right), \quad \forall \tau_{i,k} \in \Gamma_i$$

Pour minimiser le pessimisme, nous n'appliquerons la formule B.1 qu'aux transactions différentes de celles auxquelles appartient la tâche dont nous calculons le temps de réponse maximum, c'est-à-dire Γ_a. Pour cette dernière la formule B.1 reste appliquée.

Nous allons numéroter avec p, les différentes activations de $\tau_{a,b}$. $p = 1$ est la première activation après le début de la période d'activité et donc qui appartient à l'intervalle $[t_c = 0, T_i]$. L'activation, ayant lieu dans l'intervalle $[T_i, 2T_i]$, est $p = 2$ ainsi de suite pour les autres qui suivent. Dans l'intervalle $[-T_i, 0]$, c'est l'apparition $p = -1$ et de même pour les précédentes.

La date limite de fin d'exécution de la requête p de $\tau_{i,j}$ devient $\varphi_{i,j,k} + (p-1)T_i + D_{i,j}$. Ainsi, la première activation d'une requête p d'une tâche $\tau_{i,j}$ dans la période d'activité correspond à l'index :

$$p_{0,i,j,k} = -\left\lfloor \frac{J_{i,j} + \varphi_{i,j,k}}{T_i} \right\rfloor + 1$$

et la dernière :

$$p_{L,i,j,k} = \left\lceil \frac{L - \varphi_{i,j,k}}{T_i} \right\rceil$$

où L est la longueur de la période d'activité.

Si nous notons Ψ, l'ensemble des instants de la période d'activité où la date de fin d'exécution d'une requête de $\tau_{a,b}$ coïncide avec celle d'une autre requête. Alors Ψ vaut :

$$\Psi = \bigcup_{i=1..n} \bigcup_{\forall \tau_{i,j} \in \Gamma_i} \bigcup_{\forall \tau_{i,k} \in \Gamma_i} \bigcup_{\forall p = p_{0,i,j,k}..p_{L,i,j,k}} \varphi_{i,j,k} + (p-1)T_i + D_{i,j}$$

Supposons A, l'instant auquel est arrivée la première requête de $\tau_{a,b}$ dans la période d'activité. Alors, la date de fin d'exécution de la p^{ieme} requête de $\tau_{a,b}$ est notée $w^A_{a,b,c}(p)$.

$$\begin{aligned} w^A_{a,b,c}(p) = B_{a,b} + \ & (p - p_{0,a,b,c} + 1)C_{a,b} \\ & + W_{a,c}\left(w^A_{a,b,c}(p), D^A_{a,b,c}(p)\right) \\ & + \sum_{\forall i \neq a} W^*_i\left(w^A_{a,b,c}(p), D^A_{a,b,c}(p)\right) \end{aligned}$$

Où $D^A_{a,b,c}(p)$ est la date de fin d'exécution correspondant à la requête p arrivant à l'instant A :

$$D^A_{a,b,c}(p) = A + \varphi_{a,b,c} + (p-1)T_a + D_{a,b}$$

Le pire temps de réponse de la p^{ieme} requête de la $\tau_{a,b}$ est :

$$R^A_{a,b,c}(p) = w^A_{a,b,c}(p) - A - \varphi_{a,b,c}(p-1)T_a + \Phi_{a,b}$$

Pour chaque requête p, il suffit juste de vérifier les valeurs de A comprises entre 0 et T_a. Car si A est plus grand que T_a alors cela signifie que c'est une autre valeur de p. Ce qui permet de réduire Ψ à :

$$\Psi^* = \{\Psi_x \in \Psi \mid \varphi_{a,b,c} + (p-1)T_a + D_{a,b} \leq \Phi_x < \varphi_{a,b,c} + pT_a + D_{a,b}\} \tag{B.1}$$

Nous obtenons pour la longueur de la période d'activité :

$$L_{a,b,c} = W_{a,c}\left(L_{a,b,c}, \infty\right) + \sum_{\forall i \neq a} W^*_i\left(L_{a,b,c}, \infty\right)$$

Et finalement, nous obtenons pour le pire temps de réponse de $\tau_{a,b}$:

$$R_{a,b} = \max_p \left(R^A_{a,b,c}(p)\right) \ \forall p = p_{0,i,j,k} \ldots \left\lceil \frac{L-\varphi_{i,j,k}}{T_i} \right\rceil, \ \forall c \in \Gamma_a, \ \forall A \in \Psi^* \tag{B.2}$$

B.2. Systèmes de tâches à suspension

Nous réutilisons la même méthode de transformation du problème de tâches à suspension en un problème de tâches avec *offset* et gigue sur activation que dans le paragraphe A.3.

Annexe C

Test d'ordonnançabilité basé sur le calcul du facteur d'utilisation [DEV 03]

DEVI base ses travaux sur ceux de LIU [LIU 00a] présentés dans le paragraphe 5.2.6. Il reprend les résultats de LIU qui évalue à $\min(X_k, C_k)$ le délai supplémentaire infligé par une tâche τ_k sur les tâches moins prioritaires. Mais l'algorithme d'ordonnancement étant EDF, il faut adapter ce test basé sur les algorithmes à priorité fixe à EDF.

La différence notable par rapport aux algorithmes à priorité fixe, est que pour l'algorithme EDF, les priorités des tâches varient d'une requête à une autre. Ainsi, une tâche peut, au travers de ses différentes requêtes, bloquer de par sa suspension n'importe quelle autre tâche, ce qui n'est pas le cas pour les algorithmes à priorité fixe. Le retard supplémentaire subi par une requête de la tâche τ_j et dû à la suspension des autres tâches de la configuration est égal à :

$$s_j = \sum_{i \leq j \leq n \ \wedge \ i \neq j} \min(X_i, C_i) + X_j$$

Posons :

$$S_k = \sum_{i=1}^{k} \min(X_i, C_i) \tag{C.1}$$

$$S'_k = \max_{1 \leq i \leq k} \left(\max(0, X_i - C_i) \right). \tag{C.2}$$

Remarque 24 *Soit I une configuration de n tâches à suspension. Le retard maximum que subit une requête d'une tâche de I dû à l'ensemble des suspensions des tâches de la configuration I est majoré par $S_n + S'_n$ (c.-à-d. $s_j \leq S_n + S'_n, \forall j \in \{1, \ldots, n\}$).*

Le théorème suivant présente un test d'ordonnançabilité pour les tâches à suspension.

Theorème 30 *Soit I, une configuration de n tâches à suspension. Les tâches sont périodiques, non à départ simultané et préemptibles. Les tâches sont ordonnées par ordre croissant de l'échéance relative des tâches (c.-à-d. $i < j$ implique que $D_i \leq D_j$). Alors la configuration est ordonnançable par l'algorithme* EDF *si :*

$$\forall k \ : \ 1 \leq k \leq n, \ \frac{S_k + S'_k}{D_k} + \sum_{i=1}^{k} \frac{C_i}{T_i} + \frac{1}{D_k} \sum_{i=1}^{k} \left(\frac{T_i - \min(T_i, D_i)}{T_i} . C_i \right) \leq 1 \qquad \text{(C.3)}$$

où S_k (respectivement S'_k) se calcule par la formule [C.1] (respectivement [C.2]).

Exemple 25 *Exemple d'application de la méthode de* DEVI *sur la configuration de tâches à suspension I. Le tableau C.1 présente le test d'ordonnançabilité de* DEVI. *Pour déterminer l'ordonnançabilité de la configuration, il faut calculer au préalable les valeurs de S_k et S'_k pour $k \in \{1, 2, 3\}$. Puis nous calculons la valeur de la formule C.3 pour chaque k. La formule C.3 étant vérifiée pour tout k alors la configuration est ordonnançable.*

k	S_k	S'_k	formule C.3	formule C.3≤ 1?
1	1	0	$\frac{3}{8}$	oui
2	4	0	$\frac{3}{8} + \frac{2}{10}$	oui
3	6	0	$\frac{3}{8} + \frac{2}{10} + \frac{9}{80}$	oui

Tableau C.1. *Exemple d'application de la méthode de* DEVI

Annexe D

Démonstration de la garantie de performance de l'algorithme unifié

Nous commençons par donner quelques notations : d'une manière analogue à la démonstration faite pour démontrer la garantie de performance de l'algorithme en-ligne αH^{∞}, théorème 29, nous considérons le dernier bloc dans la séquence d'exécution d'une configuration par l'algorithme unifié. Par conséquent, soit I une configuration de tâches et soit $S = (B_k, \ldots, B_{m-1}, B_m)$ le dernier bloc de la séquence d'exécution de l'algorithme unifié pour l'ordonnancement de I (B_k étant le dernier lot régulier de la séquence d'ordonnancement et B_m, le dernier lot de la séquence d'exécution) (comme le représente la figure 12.4), page 178. Calculons d'abord la longueur d'ordonnancement obtenue avec l'algorithme unifié, et notée σ_{AlgU} :

Theorème 31 *La longueur d'ordonnancement obtenue avec l'algorithme unifié est de* $s_m + C_{(m)}$.

Démonstration :

Comme B_m est le dernier lot de la séquence d'exécution, il est alors clair que la longueur d'ordonnancement est égale à la date de début d'exécution du lot B_m (s_m) plus sa longueur d'exécution $C_{(m)}$:

$$\sigma_{AlgU} = s_m + C_{(m)}$$

□

Avant de déterminer la longueur d'ordonnancement de l'algorithme optimal, démontrons les résultats des trois lemmes suivants :

Lemme 9 *Pour tout i, avec $k \leq i \leq m$ la propriété suivante est vérifiée :*

$$s_i \geq \alpha C_{(i)}$$

Démonstration :

Pour démontrer ce lemme, il suffit de remarquer que l'exécution de chaque lot B_i ($k \leq i \leq m-$) commence après le délai d'attente de longueur minimale $\alpha C_{(i)}$, ce qui nous permet de conclure :

$$s_i \geq \alpha C_{(i)}$$

□

Lemme 10 *Pour tout i, avec $k \leq i \leq m-1$ la propriété suivante est vérifiée :*

$$r_{(i+1)} \geq \alpha s_i$$

Démonstration :

L'ensemble des tâches du lot B_{i+1} avec $k \leq i \leq m-1$ est arrivé après le lancement de l'exécution du lot B_i sinon il serait exécuté dans le lot B_i. En particulier c'est vrai pour la tâche $\tau_{(i+1)}$, d'où le résultat.

□

Lemme 11 *Pour le lot B_k,*

$$s_k \geq (1+\alpha) r_{(k)} + \alpha C_{(k)}$$

Démonstration :

Le lot B_k est régulier, il a été ordonnancé juste à la fin de son délai d'attente. Or ce délai est inférieur à $(1+\alpha) r_{(k)} + \alpha C_{(k)}$, d'où le résultat.

□

Theorème 32 *Si le dernier bloc de la séquence d'exécution par l'algorithme unifié s'écrit* $S = (B_k, \ldots, B_{m-1}, B_m)$ *alors la longueur d'ordonnancement obtenue par l'algorithme optimal, notée* $\sigma^*(S)$ *est de* $\sigma^*(S) \geq \alpha(s_m + C_{(m)})$.

Démonstration :

La preuve de ce théorème se fait par induction sur le nombre de lots composant le dernier bloc dans la séquence d'exécution par l'algorithme unifié. Le dernier bloc s'écrit : $(B_k, \ldots, B_{m-1}, B_m)$.

Supposons qu'il n'y ait qu'un seul lot dans le dernier bloc : $k = m$, $S = (B_m)$. Dans ce cas, l'algorithme optimal doit ordonnancer toutes les tâches de la configuration et en particulier la tâche $\tau_{(m)}$ (tâche ayant le plus grand temps processeur et étant ordonnancée dans le lot B_m). Dans ce cas :

$$\begin{aligned}
\sigma^*(S) &\geq r_{(m)} + C_{(m)} \\
&\geq \alpha(1+\alpha)(r_{(m)} + C_{(m)}) \\
&\geq \alpha((1+\alpha)r_{(m)} + C_{(m)} + \alpha C_{(m)}) \\
&\geq \alpha(s_m + C_{(m)})
\end{aligned}$$

Cette inégalité est obtenue en utilisant le lemme 11 avec $k = m$.

Nous supposons maintenant que si le dernier bloc est constitué des lots :
$S' = (B_l, \ldots, B_{m-1}, B_m)$ avec $l > k$ et si les lots de ce bloc vérifient les lemmes 9, 10 et 11 alors $\sigma^*(S') \geq \alpha(s_m + C_{(m)})$. Maintenant, montrons que $\sigma^*(S) \geq \alpha(s_m + C_{(m)})$ où $S = (B_k, \ldots, B_{m-1}, B_m)$. Trois cas sont à étudier :

Premier cas : Il existe j tel que $k + 1 \leq j \leq m$ et $C_{(j-1)} \leq C_{(j)}$.
Soit $S' = (B_j, \ldots, B_m)$ et nous montrons que les lots du bloc S' vérifient les lemmes 9, 10 et 11. Pour les lemmes 9 et 10, c'est une conséquence de la conception du bloc S. Pour le lemme 11, il suffit d'observer que le lot B_j est retardé donc, en utilisant la propriété 5 :

$$\begin{aligned}
s_j &\geq s_{j-1} + C_{(j-1)} \\
&\geq s_{j-1} + \alpha(C_{(j-1)} + \alpha C_{(j-1)})
\end{aligned}$$

Maintenant, en utilisant le fait que $C_{(j-1)} \leq C_{(j)}$ et le lemme 9, nous obtenons :

$$\begin{aligned}
s_j &\geq s_{j-1} + \alpha(C_{(j)} + s_{j-1}) \\
&\geq (1+\alpha)r_{(j)} + \alpha C_{(j)}
\end{aligned}$$

La dernière ligne est établie en utilisant le lemme 10.

Ainsi, les lemmes 9, 10 et 11 sont vérifiés pour S' et donc par induction,
$\sigma^*(S') \geq \alpha(s_m + C_{(m)})$. Or $\sigma^*(S) \geq \sigma^*(S) \geq \alpha(s_m + C_{(m)})$.

Second cas : Les temps processeurs des tâches, les plus longues de chaque lot, satisfont : $C_{(k)} > \cdots > C_{(m)}$ et il existe j tel que $k \leq j < m$ et la tâche $\tau_{(m)}$ est ordonnancée par l'algorithme optimal dans le même lot que la tâche $\tau_{(j)}$.
Alors, après l'activation de la tâche $\tau_{(m)}$ (à l'instant $r_{(m)}$), un lot a été lancé par l'algorithme optimal et de longueur au moins $C_{(j)}$. Ainsi, en utilisant le résultat du lemme 10,

$$\begin{aligned} \sigma^*(S) &\geq r_{(m)} + C_{(j)} \\ &\geq s_{m-1} + C_{(j)} \\ &\geq \alpha s_{m-1} + (1-\alpha)s_{m-1} + C_{(j)} \end{aligned}$$

Maintenant, en utilisant le lemme 9, le fait que $C_{(j)} \geq C_{(m-1)}$ et finalement la propriété 5 nous obtenons :

$$\begin{aligned} \sigma^*(S) &\geq \alpha s_{m-1} + (1-\alpha)\alpha C_{(m-1)} + C_{(m-1)} \\ &\geq \alpha s_{m-1} + 2C_{(m-1)} \end{aligned}$$

Enfin, comme le lot B_m est retardé, $s_m = s_{m-1} + C_{(m-1)}$ et comme $C_{(m-1)} > C_{(m)}$, donc :

$$\sigma^*(S) \geq \alpha(s_m + C_{(m)})$$

Troisième cas : Les temps processeurs des tâches, les plus longues de chaque lot, satisfont : $C_{(k)} > \cdots > C_{(m)}$ et la tâche $\tau_{(m)}$ n'est pas ordonnancée par l'algorithme optimal avec une autre tâche des lots B_j avec $k \leq j \leq m-1$.

D'après le lemme 10, $r_{(m)}$ est supérieure à s_{m-1}. Donc dans l'ordonnancement obtenu par l'algorithme optimal et d'après l'hypothèse de départ, il faut ordonnancer les tâches des lots $B_k, \ldots, B_{m-1}$ puis un lot avec la tâche $\tau_{(m)}$. Ainsi, si nous notons $S' = (B_k, \ldots, B_{m-1})$ alors $\sigma^*(S) \geq \sigma^*(S') + C_{(m)}$. À noter que les lots du bloc S' vérifient les lemmes 9, 10 et 11 par construction de S donc par induction, $\sigma^*(S') \geq \alpha(s_{m-1} + C_{(m-1)})$. Et en utilisant le fait que le lot B_m est en retard,

$$\begin{aligned} \sigma^*(S) &\geq \alpha(s_{m-1} + C_{(m-1)}) + C_{(m)} \\ &\geq \alpha s_m + C_{(m)} \\ &\geq \alpha(s_m + C_{(m)}) \end{aligned}$$

En conclusion, la longueur d'ordonnancement obtenue par l'algorithme optimal est de $\sigma^*(S) \geq \alpha(s_m + C_{(m)})$ si le dernier bloc de la séquence d'exécution par l'algorithme unifié s'écrit $S = (B_k, \ldots, B_{m-1}, B_m)$.

Theorème 33 *La garantie de performance de l'algorithme unifié, pour le problème d'ordonnancement* $1|p - batch, r_i, b = \infty|C_{max}$ *est de* $\frac{1+\sqrt{5}}{2}$.

Démonstration : Cette démonstration se base sur les résultats des théorèmes 31 et 32. Ainsi, soit I une configuration de tâches et $S = (B_k, \ldots, B_{m-1}, B_m)$ le dernier bloc dans l'ordonnancement obtenu par l'algorithme unifié. Alors, la performance de l'algorithme unifié pour l'ordonnancement de I est égale à : $\sigma_{AlgU}(I) = s_m + C_{(m)}$ (*cf.* théorème 31) et celle de l'algorithme optimal : $\sigma^*(I) = \sigma^*(S) \geq \alpha(s_m + C_{(m)})$(*cf.* théorème 31). Par conséquent, nous obtenons un ratio de compétitivité c_{AlgU} pour l'algorithme unifié de :

$$\begin{aligned} c_{AlgU} = \quad &\geq \quad \sup_{pour\ toute\ configuration\ I} \frac{\sigma_{AlgU}(I)}{\sigma^*(I)} \\ &\geq \quad \frac{\sigma_{AlgU}(I)}{\sigma^*(I)} \\ &\geq \quad \frac{s_m + C_{(m)}}{\alpha(s_m + C_{(m)})} \\ &\geq \quad \frac{1}{\alpha} \\ &\geq \quad 1 + \alpha \end{aligned}$$

□

Bibliographie

Bibliographie

[ABE 98] ABENI L., « Server Mechanisms for Multimedia Applications », *Technical Report RETIS TR98-0101, Scuola Superiore S.Anna, Pisa, Italy*, 1998.

[AKK 00] VAN DEN AKKER M., HOOGEVEEN H., VAKHANIA N., « Restarts can help in the on-line minimization of the maximum delivery time on a single machine », *proc. European Symposium on Algorithms*, p. 427-436, 2000.

[ALB 04] ALBERS K., SLOMKA F., « An event stream driven approximation for the analysis of real-time systems », *Euromicro Int. Conf. on Real-Time Systems (ECRTS'04)*, 2004.

[ALZ 05] ALZEER I., MOLINARO P., TRINQUET Y., « Calcul exhaustif du Temps de Réponse de tâches et messages dans un système temps réel réparti », *In Proceedings of the* 13^th *Real-Time Systems*, 2005.

[AND 02] ANDERSON E., POTTS C., « On-line scheduling of a single machine to minimize total weighted completion time », *in : proc. ACM-SIAM Symposium on Discrete Algorithms*, p. 548-557, 2002.

[AND 03] ANDERSSON B., Static-priority scheduling on multiprocessors, PhD thesis, Chalmers University of Technology, Göteborg, Sweden, 2003.

[AUD 93] AUDSLEY N., BURNS A., RICHARDSON M., TINDELL K., WELLINGS A., « Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling », *Software Engineering Journal*, vol. 8, n°5, p. 284-292, 1993.

[BAK 74] BAKER K., *Introduction to sequencing and scheduling*, John Wiley & Sons, New-York, 1974.

[BAK 91] BAKER T., « Stack-based scheduling of real-time processes », *Journal of Real-Time Systems*, vol. 3, n°1, p. 67-99, mars 1991.

[BAP 00] BAPTISTE P., « Batching Identical Jobs », *Mathematics methods of Operations Research*, vol. 53, p. 355-367, 2000.

[BAR 90a] BARUAH S., MOK A., ROSIER L., « Preemptively scheduling hard-real-time sporadic tasks on one processor », *In Proceedings of the* 11^{th} *IEEE Real-Time Systems Symposium (RTSS'90)*, p. 182-190, 1990.

[BAR 90b] BARUAH S., ROSIER L., HOWELL R., « Algorithms and Complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor », *Journal of Real-Time Systems*, vol. 1, p. 301-324, 1990.

[BAR 94] BARUAH S., HARITSA J., SHARMA N., « On-line scheduling to maximise task completions », *in : proc. Real-Time Systems Symposium*, p. 228-236, 1994.

[BAR 01] BARUAH S., HARITSA J., SHARMA N., « On-line scheduling to maximise task completions », *The Journal of Combinatorial Mathematics and Combinatorial Computing*, vol. 39, p. 65-78, 2001.

[BAR 03] BARUAH S., « Dynamic- and static-priority scheduling of recurring real-time tasks », *Journal of Real-Time Systems*, vol. 24, n°1, p. 93-128, 2003.

[BAR 04] BARUAH S., GOOSSENS J., « Scheduling Real-time Tasks : Algorithms and Complexity », *In Handbook of Scheduling : Algorithms, Models, and Performance Analysis, Joseph Y-T Leung (edt). Chapman Hall/CRC Press*, 2004.

[BIN 03] BINI E., BUTTAZZO G., BUTTAZZO G., « Rate Monotonic Analysis : The Hyperbolic Bound », *IEEE Transactions On Computers*, vol. 52, n°7, page , juillet 2003.

[BIN 04a] BINI E., BUTTAZZO G., « Biasing Effects in Schedulability Measures », *Proceedings of the 16th Euromicro Conference on Real-Time Systems, Catania, Italy*, page , June 2004.

[BIN 04b] BINI E., BUTTAZZO G., « Schedulability Analysis of Periodic Fixed-Priority Systems », *IEEE Transactions on Computers*, vol. 53, n°11, page , novembre 2004.

[BLA 96] BLAZEWICZ J., ECKER K., PESCH E., SCHMIDT G., WEGLARZ J., *Scheduling in Computer and Manufacturing Systems*, Springer Verlag, Berlin, 1996.

[BOR 98] BORODIN A., R.EL-YANIV, *Online Computation and Competitive analysis*, Cambridge University Press, 1998.

[BRI 06] BRIL R., « Existing Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Refuted », *Work-In-Progress Session of the 18th Euromicro Conference on Real-Time Systems*, vol. 1, n°5–7, p. 1–4, Juillet 2006.

[BRU 98] BRUCKER P., GLADKY A., HOOGOVEEN H., KOVALYOV M., POTTS C., TAUTENHAHN T., VANDEVELDE S., « Scheduling a batching machine », *Journal of Scheduling*, vol. 1, n°1, p. 31-58, 1998.

[BRU 01] BRUCKER P., *Scheduling Algorithms*, (Third edition) Springer Verlag, 2001.

[BUR 95] BURNS A., « Preemptive Priority-Based Scheduling : An Appropriate Engineering Approach », *in Advances in Real-Time Systems, S.H. Son, Ed., Prentice Hall, New Jersey*, p. 225-248, 1995.

[CAR 88] CARLIER J., CHRÉTIENNE P., *Problèmes d'ordonnancement, modélisation, complexité, algorithmes*, Masson, 1988.

[CHA 05] CHAN W., LAM T., LIU K., WONG W., « New Resource Augmentation Analysis of the Total Stretch of SRPT and SJF in Multiprocessor Scheduling », *30th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, page , 2005.

[CHE 89] CHETTO H., CHETTO M., « Some results of the earliest deadline scheduling algorithm », *IEEE Transitions on Software Engineering*, vol. 15, n°10, p. 1261–1269, 1989.

[CHE 90] CHETTO H., SILLY M., BOUCHENTOUF T., « Dynamic scheduling of real-time task under precedence constraints », *Real Time Systems*, vol. 2, n°2, p. 181-194, 1990.

[CHE 96] CHEN B., VESTJENS A., « Scheduling on identical machines : How good is LPT in an on-line setting ? », *Operations Research Letters*, , n°13, March 1996.

[CHE 01] CHENG T., LIU Z., YU W., « Scheduling jobs with release dates and deadlines on a batch processing machines », *IIE Transactions*, vol. 33, p. 685-690, 2001.

[CHE 04] CHENG T., YAN J., YANG A., « Scheduling a batch-processing machine subject to precedence constraints, release dates and identical processing times », *Computer and Operations Research*, vol. 32, p. 849-859, 2004.

[CNR 88] CNRS, « Le temps réel - Groupe de réflexion sur le temps réel », *Techniques et Sciences Informatiques*, vol. 8, n°5, p. 493–500, 1988.

[CON 67] CONWAY R., MAXWELL W., MILLER L., *Theory of Scheduling*, Addison Wesley, Reading, Mass., USA, 1967.

[COT 00] COTTET F., DELACROIX J., KAISER C., MAMMERI Z., *Ordonnancement temps réel*, Hermès Editions, 2000.

[COT 05] COTTET F., GROLLEAU E., *Systèmes temps réel de Contrôle-Commande, conception et implémentation*, Dunod, Paris, 2005.

[DEN 99] DENG X., POON C. K., ZHANG Y., « Approximation Algorithms in Batch Processing », vol. LNCS 1741, p. 153-162, December 1999.

[DER 74] DERTOUZOS M., « Control robotics : the procedural control of physical processors », *IFIP Congress*, p. 807-813, 1974.

[DER 89] DERTOUZOS M., MOK A., « Multiprocessor on-line scheduling of hard real-time tasks », *IEEE Trasnactions on Software Engineering*, vol. 15, p. 1497-1506, 1989.

[DEV 00] DEVILLERS R., GOOSSENS J., « Liu and Layland's schedulability test revisited », *Information Processing Letters*, vol. 73, n°5-6, p. 157-161, March 2000.

[DEV 03] DEVI U., « An Improved Schedulability Test for Uniprocessor Periodic Task Systems », *IEEE Euromicro Conf. on Real-Time Systems (ECRTS'03)*, p. 23-32, 2003.

[DOR 91] DORSEUIL A., P. PILLOT, *Le temps réel en milieu industriel : Concepts, environnements, multitâches*, Dunod, Paris, 1991.

[DUP 02] DUPONT L., DHAENENS-FLIPO C., « Minimizing the makespan on a batch machine with non-identical job sizes : an exact procedure », *Computer and Operations Research*, vol. 29, n°7, p. 807-819, 2002.

[E.G 76] E.G. COFFMAN J., *Scheduling in Computer and Job Shop Systems*, Wiley, New York, 1976.

[EPS 01] EPSTEIN L., VAN-STEE R., « Lower bounds for on-line single machine scheduling », *proc. 26th Mathematical Foundations of Computer Science*, p. 338-350, 2001.

[EPS 03] EPSTEIN L., VAN-STEE R., « Lower bounds for on-line single-machine scheduling », *Theoretical Computer Science*, , n°299, p. 439-450, 2003.

[ESQ 99] ESQUIROL P., LOPEZ P., *L'ordonnancement*, Economica, Paris, 1999.

[FIS 05] FISHER N., BARUAH S., « A fully polynomial-time approximation scheme for feasibility analysis in static-priority systems with arbitrary relative deadlines », *proc. Proceedings of the EuroMicro Conference on Real-Time Systems, (ECRTS'05)*, 2005.

[GAR 79] GAREY M., JOHNSON D., *Computers and Intractability : a guide to the theory of NP-Completeness*, W.H. Freeman, San Francisco, 1979.

[GRA 69] GRAHAM R., « Bounds on multiprocessing timing anomalies », *SIAM Journal of Allied Mathematics*, vol. 17, n°2, p. 416-429, décembre 1969.

[GRA 96] GRAHAM R., LAWLER E., LENSTRA J., KAN A. R., « Optimisation and approximation in deterministic sequencing and scheduling : A survey », *Annals of Discrete Mathematics*, vol. 5, p. 287-326, 1996.

[GUT 98] GUTIÉRREZ J. P., GARCÍA J. J. G., GONZÁLEZ-HARBOUR M., « Best-Case Analysis for Improving the Worst-Case Schedulability Test for Distributed Hard Real-Time Systems », *10th Euromicro Workshop on Real-Time Systems, Berlin, Germany*, juin 1998.

[HAR 85] HAREL D., PNUELLI A., « On the development of Reactive Systems, in Logic and Models of Concurrent Systems », *NATO ASI in Computer Science*, p. 447-498, 1985.

[HOO 96] HOOGEVEEN H., VESTJENS A., « Optimal on-line algorithms for single-machine scheduling », *in : proc. 5th Conference on Integer Programming and Combinatorial Optimization*, p. 404-414, 1996.

[HOO 00] HOOGEVEEN H., POTTS C. N., WOEGINGER G. J., « On-line scheduling on a single machine : maximizing the number of early jobs », *Operations Research Letters*, vol. 27, p. 193-197, 2000.

[IKU 86] IKURA Y., GIMPLE M., « Efficient scheduling algorithms for a single batch processing machine », *Operations Research Letters*, vol. 5, p. 61-65, 1986.

[JAC 55] JACKSON J., « Scheduling a production line to minimize maximum tardiness », *Management Science Research Project 43, UCLA*, janvier 1955.

[JEF 91] JEFFAY K., STANAT D., MARTEL C., « On Non-Preemptive Scheduling of Periodic and Sporadic tasks », *proc. Real-Time Systems Symposium*, p. 129-139, 1991.

[JEF 92] JEFFAY K., « Scheduling sporadic tasks with shared resources in hard-real-time systems », *In Proceeding of the* 13^{th} *IEEE Real-Time Systems Symposium*, p. 89-99, 1992.

[JOS 86] JOSEPH M., PANDYA P., « Finding response times in real-time systems », *Comp. Journal*, vol. 29, n°5, page , 1986.

[KAL 95] KALYANASUNDARAM B., PRUHS K., « Speed is as powerful as clairvoyance », *proc. IEEE Foundations of Computer Science*, p. 214-223, 1995.

[KAR 92] KARP R., « On-line algorithms versus off-line algorithms : How much is it worth to know the future ? », *Algorithms, Software, Architecture, IFIP Transactions A-12,Vol. Information Processing*, vol. 1, p. 416-429, 1992.

[KIM 95] KIM I., CHOI K., PARK S., KIM D., HONG M., « Real-Time Scheduling of tasks that contain the external blocking intervals », *proc. Conference on Real-Time Computing Systems and Applications*, p. 54-59, 1995.

[KLE 93] KLEIN M., RALYA T., POLLAK B., OBENZA R., GONZALES-HARBOUR M., *A practitioner's handbook for Real Time Systems Analysis*, Kluwer Academic Publishers, 1993.

[KOP 83] KOPETZ H., « Real-time in Distributed Real-Time Systems », *Real-time in distributed real-time systems. Proc. 5th IFAC workshop on Distributed Computer Control Systems, Oxford, UK*, 1983.

[LAB 74] LABETOULLE J., « Un algorithme optimal pour la gestion des processus en temps réel », *Revue Française d'Automatique, Informatique et Recherche Opérationnelle*, p. 11-17, janvier 1974.

[LAP 91] LAPLANTE P. A., « Real-Time Systems Design and Analysis, an Engineer's handbook », *IEEE Computer Society Press*, vol. 0-8186-3107-4, 1991.

[LAW 93] LAWLER E., LENSTRA J., KAN A. R., SHMOYS D., « Sequencing and scheduling : algorithms and complexity », *in : Handbooks in Operations Research and Management Science. North Holland*, vol. 4, p. 445-552, 1993.

[LEE 99] LEE C., UZSOY R., « Minimizing makespan on a single batch processing machine with dynamic job arrivals », *International Journal of Production Research*, vol. 37, n°1, p. 219-236, 1999.

[LEH 87] LEHOCZKY J., SHA L., STROSNIDER J., « Enhanced Aperiodic Responsiveness in Hard Real-Time Environments », *Proceedings 8th IEEE Real-Time System Symposium, San Jose, California*, p. 261-270, décembre 1987.

[LEH 89] LEHOCZKY J., SHA L., DING Y., « The Rate Monotonic Scheduling Algorithm : Exact Characterization and Average Case behavior », *In Proceedings of the* 10^{th} *IEEE Real-Time Systems Symposium (RTSS'89)*, p. 166-171, 1989.

[LEN 77] LENSTRA J., « Sequencing by enumerative methods », *Mathematical Centre Tracts*, vol. 69, page , 1977.

[LEU 80] LEUNG J., MERRILL M., « A note on preemptive scheduling of periodic, real-time tasks », *Information processing letters*, vol. 11, n°3, p. 115-118, 1980.

[LEU 82] LEUNG J., WHITEHEAD J., « On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks », *perf. Eval. (Netherlands)*, vol. 2, p. 237-250, 1982.

[LIU 73] LIU C., LAYLAND J., « Scheduling algorithms for multiprogramming in a hard real-time environment », *Journal of the ACM*, vol. 20, n°1, p. 40-61, 1973.

[LIU 00a] LIU J., *Real-Time Systems*, Prentice hall, 2000.

[LIU 00b] LIU Z., YU W., « Scheduling one batch processor subject to job release dates », *Discrete Applied Mathematics*, vol. 105, p. 129-136, 2000.

[MAN 98] MANABE Y., AOYAGI S., « A feasibility decision algorithm for Rate Monotonic and Deadline Monotonic scheduling », *Journal of Real-Time Systems*, vol. 14, n°2, p. 171-181, 1998.

[MIN 94] MING L., « Scheduling of the Inter-Dependent Messages in Real-Time Communication », *Proc. of the First International Workshop on REal-Time Computing Systems and Applications*, décembre 1994.

[MOK 83] MOK A., *Fundamental design problems of distributed systems for the hard real-time environment*, PhD Thesis, Massachusetts Institute of Technology, 1983.

[MOK 05] MOK A., POON W.-C., « Non-Preemptive Robustness under Reduced System Load », 26^{th} *IEEE International Real-Time Systems Symposium (RTSS'05)*, vol. 1, n°5-8, p. 200-209, December 2005.

[NAV 06] NAVET N., Ed., *Systèmes temps réel — Ordonnancement, réseaux et qualité de service*, vol. 2, Hermès, 2006, Chapitre 2 : Ordonnancement temps réel multiprocesseur (24 pages). ISBN 2–7462–1304–4.

[PAL 98] PALENCIA J., GONZALES-HARBOUR M., « Schedulability Analysis for Tasks with Static and Dynamic Offsets », *Proceedings of the* 19^th *IEEE Real-Time Systems Symposium*, 1998.

[PAL 03] PALENCIA J., GONZALES-HARBOUR M., « Offset-Based Response Time Analysis of Distributed Systems Scheduled under EDF », *proc IEEE Euromicro Conf. on Real-Time Systems (ECRTS'03)*, page , 2003.

[PAR 04] PARK M., CHO Y., « Feasibility analysis of hard real-time periodic tasks », *Journal of Systems and Softwares*, vol. 73, p. 89-100, 2004.

[PHI 95] PHILLIPS C., STEIN C., WEIN J., « Scheduling jobs that arrive over time », *in : proc. 4th Workshop on Algorithms and Data Structures, LNCS 955, Springer Verlag*, p. 86-97, 1995.

[PHI 97] PHILIPS C., STEIN C., TORNG E., WEIN J., « Optimal time-critical scheduling via resource augmentation », *prc.* 29^{th} *Ann. ACM Symp. on Theory of Computing*, p. 110–149, 1997.

[PIN 01] PINEDO M., *Scheduling : Theory, Algorithms, and Systems (2nd Edition)*, Prentice Hall, Englewood Cliffs, 2001.

[POO 00] POON C., ZHANG P., « Minimizing makespan in Batch machine Scheduling », *in : proc Int. Symposium on Algorithms and Computations (ISAAC 2000), Lecture Notes in Computer Science, Springer Verlag*, , n°LNCS 1969, p. 386-397, 2000.

[POO 05a] POON C., YU W., « A flexible on-line scheduling algorithms for batch machine with infinite capacity », *Annals of Operations Research*, vol. 133, p. 175-181, 2005.

[POO 05b] POON C., YU W., « On-line scheduling algorithms for a batch machine with finite capacity », *Journal of Combinatorial Optimization*, vol. 9, n°2, p. 167-186, 2005.

[PRU 04] PRUH K., SGALL J., TORNG E., « *Handbook of Scheduling : Algorithms, Models, and Performance Analysis* », vol. 2, Chapitre Online scheduling, Editor : Joseph Y-T. Leung, CRC Press, Boca Raton, FL, USA, 2004.

[RAJ 89] RAJKUMAR R., « Task synchronization in real-time systems », *PhD Thesis, Carnegie Mellon University,*, août 1989.

[RAJ 91] RAJKUMAR R., « Synchronisation in Real-Time Systems : A priority Inheritance Approach », *Kluwer Academic Publishers*, 1991.

[RIC 01] RICHARD P., COTTET F., RICHARD M., « On-line scheduling of Real-Time Distributed Computers With Complex Communication Constraints », *ICECCS'2001, edited by Press, IEEE Computer, Skövde (Sweden)*, p. 26-34, 2001.

[RIC 02] RICHARD M., RICHARD P., GROLLEAU E., COTTET F., « Contraintes de précédences et ordonnancement monoprocesseur », *proc. Embedded Systems (RTS'02), Paris*, , n°1, p. 121-138, 2002.

[RIC 03a] RICHARD P., « On the complexity of scheduling real-time tasks with self-suspensions on one processor », *IEEE Euromicro Conf. on Real-Time Systems (ECRTS'03)*, page 8p, 2003.

[RIC 03b] RICHARD P., RIDOUARD F., MARTINEAU P., « On-line scheduling on a single batching machine to minimize the makespan », 6^{th} *Int. Conference on Industrial Engineering and Production Management (IEPM'03), Porto (Portugal)*, , n°26-28, may 2003.

[RID 03] RIDOUARD F., RICHARD P., COTTET F., « Algorithmes d'ordonnancement en-ligne à une machine », *École d'Automne de Recherche Opérationnelle, Tours*, vol. 1, n°28–31, octobre 2003.

[RID 04a] RIDOUARD F., RICHARD P., COTTET F., « Negative results for scheduling independent hard real-time tasks with self-suspensions », 25^{th} *IEEE International Real-Time Systems Symposium (RTSS'04)*, 2004.

[RID 04b] RIDOUARD F., RICHARD P., MARTINEAU P., « On-line minimization of makespan for single batching machine scheduling problems », 9^{th} *International Workshop on Project Management and Scheduling (PMS'04)*, p. 287-290, 2004.

[RID 05] RIDOUARD F., RICHARD P., COTTET F., « Ordonnancement des tâches indépendantes avec suspension », *proc. Real-Time Embedded Systems (RTS'05), Paris*, 2005.

[RID 06a] RIDOUARD F., RICHARD P., « Worst-case analysis of feasibility tests for self-suspending tasks », *proc. International Conference on Real-Time and Network Systems (RTNS 2006), Poitiers*, vol. 1, n°30-31, p. 15–24, mai

[RID 06b] RIDOUARD F., RICHARD P., COTTET F., TRAORE K., « Some results on scheduling tasks with self-suspensions », *Journal of Embedded Computing (JEC), to appear*, 2006.

[RID 06c] RIDOUARD F., RICHARD P., MARTINEAU P., « On-line scheduling on a batch with unbounded batch size to minimize the makespan », *European Journal of Operational Research (EJOR), to appear*, 2006.

[RIP 96] RIPOLL I., CRESPO A., MOK A., « Improvement in feasibility testing for real-time tasks », *Journal of Real-Time Systems*, vol. 11, n°1, p. 19-39, 1996.

[SGA 98] SGALL J., « On-line scheduling - a survey », *In A. Fiat and G. Woeginger, editors, On-Line Algorithms : The State of the Art, Lecture Notes in Computer Science, Springer Verlag*, vol. 1442, p. 196-231, 1998.

[SHA 88] SHA L., GOODENOUGH J., RALYA T., « An analytic approach to real-time software engineering », *Softw. Engin. Inst. Draft Report*, 1988.

[SHA 90] SHA L., RAJKUMAR R., LEHOCZKY J., « Priority inheritance protocols : An approach to real-time system synchronization », *IEEE Transactions on Computers*, vol. 39, n°9, p. 1175-1189, 1990.

[SHA 04] SHA L., ABDELZAHER T., ARZÈN K., CERVIN A., BAKER T., BURNS A., BUTTAZZO G., LEHOCZKY J., MOK A., « Real-Time Scheduling Theory : A Historical Perspective », *Journal of Real-Time Systems*, vol. 28, n°2, p. 101-156, décembre 2004.

[SIM 05] SIMONOT-LION F., SONG Y., *Design and validation process of in-vehicle embedded electronic systems, in The Embedded Systems Handbook*, Richard Zurawski (Edt), CRC Press - Taylor & Francis, New-York, 2005.

[SLE 85] SLEATOR D. D., TARJAN R. E., « Amortized efficiency of list update and paging rules », *Communication of the ACM 28*, vol. 2, p. 202-208, 1985.

[SPR 88] SPRUNT B., LEHOCZKY J., SHA L., « Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm », *Proceedings 9th IEEE Real-Time System Symposium, Hunstville*, p. 251-258, décembre 1988.

[SPR 89] SPRUNT B., SHA L., LEHOCZKY J., « Aperiodic Task Scheduling for Hard Real-Time Systems », *The Journal of Real-Time Systems*, p. 27-69, 1989.

[SPU 93] SPURI M., STANKOVIC J., « How to integrate precedence constraints and shared resources in real-time scheduling », *Technical Report UM-CS-1993-019, U. Mass*, 1993.

[SPU 94] SPURI M., BUTTAZZO G., « Efficient Aperiodic Service under Earliest Deadline Scheduling », *In Proceedings of the* 15^{th} *IEEE Real-Time Systems Symposium (RTSS'94), San Juan, Puerto Rico*, p. 2-11, décembre 1994.

[SPU 95] SPURI M., BUTTAZZO G., SENSINI F., « Robust Aperiodic Scheduling under Dynamic Priority Systems », *In Proceedings of the* 16^{th} *IEEE Real-Time Systems Symposium (RTSS'95), Pisa, Italy,*, p. 210-219, décembre 1995.

[SPU 96a] SPURI M., « Analysis of deadline scheduled real-time systems », *INRIA Reseach Report 2772*, page34p, 1996.

[SPU 96b] SPURI M., BUTTAZZO G., « Scheduling Aperiodic Tasks in Dynamic Priority Systems », *Journal of Real-Time Systems*, vol. 10, n°2, p. 179-210, mars 1996.

[STA 88] STANKOVIC J. A., « Misconception about Real-Time Systems - a serious problem for next generation systems », *IEEE Computer*, p. 10–19, October 1988.

[STA 95] STANKOVIC J., SPURI M., DINATALE M., BUTTAZZO G., « Implications of classical scheduling results for real-time systems », *IEEE Computer*, vol. 28, n°6, p. 16-25, 1995.

[STR 95] STROSNIDER J., LEHOCZKY J., SHA L., « Algorithms and Complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor », *IEEE Transactions on Computers*, vol. 44, n°I, janvier 1995.

[SUN 00] SUNG C., CHOUNG Y., « Minimizing makespan on a single burn-in oven in semiconductor manufacturing », *European Journal of Oerational Research*, vol. 120, p. 559-574, 2000.

[TIN 92] TINDELL K., « An extendible approach for analyzing fixed priority hard real-time tasks », *Technical Report YCS189*, 1992.

[TIN 94] TINDELL K., CLARK J., « Holistic schedulability analysis for distributed real-time systems », *Microprocessors and Microprogramming*, page , 1994.

[TOV 02] TOVEY A., « Tutorial on computational complexity », *Interface*, vol. 32, n°3, p. 30-61, Mai-Juin 2002.

[TRI 05] TRINQUET Y., « Les systèmes d'exploitation temps réel », 4^{ieme} *École d'été temps réel, Nancy*, vol. 1, n°13–16, p. 123–134, septembre 2005.

[VAN 02] VAN-STEE R., POUTRÉ J. L., « Minimizing total completion time on-line on a single machine, using restarts », *10th European Symposium on Algorithms, Lecture Notes in Computer Science, Springer Verlag*, 2002.

[ZHA 01] ZHANG G., CAI X., WONG C., « On-line algorithms for minimizing makespan on batch processing machines », *Naval Research Logistics*, vol. 48, p. 241-258, 2001.

Glossaire

Glossaire

~ $\mathcal{A}$ ~

Analyse de compétitivité : technique d'analyse et d'évaluation des algorithmes en-ligne dans leur pire cas d'exécution.

Algorithme d'ordonnancement : algorithme qui organise l'exécution des tâches sur un porcesseur en veillant àce qu'elles respectent leurs contraintes temporelles tout en optimisant éventuellement certains critères.

Algorithme d'ordonnancement à priorités dynamiques : algorithme qui à chaque instant peut modifier son allocation de priorités aux tâches.

Algorithme d'ordonnancement à priorités fixes pour les requêtes : algorithme qui alloue une unique priorité pour toute la durée de l'ordonnancement d'une requête.

Algorithme d'ordonnancement à priorités fixes pour les tâches : algorithme qui choisit une unique priorité par tâche pour tout l'ordonnancement.

Algorithme d'ordonnancement clairvoyant : algorithme qui connaît les caractéristiques de toutes les tâches à venir.

Algorithme d'ordonnancement conservatif : algorithme qui exécute une tâche dès qu'elle est disponible et que le processeur est inoccupé.

Algorithme d'ordonnancement déterministe : algorithme qui prend ses décisions d'ordonnancement sans aléatoire.

Algorithme d'ordonnancement en-ligne : algorithme qui ne connaît à chaque instant de l'ordonnancement que l'ensemble des tâches qui sont déjà arrivées.

Algorithme d'ordonnancement hors-ligne : algorithme qui connaît les caractéristiques de toutes les tâches avant même de commencer l'ordonnancement.

Algorithme d'ordonnancement optimal : un algorithme est optimal pour un problème d'ordonnancement, si toute instance du problème qui est ordonnançable par un algorithme alors elle l'est aussi par cet algorithme et in versement.

Algorithme d'ordonnancement randomisé : un algorithme est dit randomisé si ses choix d'ordonnancement sont conduits tout ou partie par un choix aléatoire.

Anomalie d'ordonnancement : une anomalie survient dès lors que diminuer la durée d'exécution d'une tâche peut augmenter le temps de réponse de la tâche ou d'une autre.

Architecture logicielle : elle est composée d'un exécutif temps réel et du programme à exécuter sur l'exécutif (représenté par un ensemble de tâches).

Architecture matérielle : composants physiques sur lesquelles l'application s'appuie.

$\sim \mathcal{C} \sim$

Configuration de tâches : ensemble de tâches à ordonnancer.

Contrainte de précédence : une contrainte d'exécution existe entre la tâche τ_i et la tâche τ_j si la tâche τ_j doit attendre la fin d'exécution de la tâche τ_i pour s'exécuter.

$\sim \mathcal{D} \sim$

Date d'activation (d'une tâche) : instant de réveil de la tâche dans le système

Délai de suspension (d'une tâche) : durée de suspension d'une tâche au court de son exécution.

Demande processeur (Analyse) : calcul de l'occupation du processeur sur des intervalles d'exécution.

Durée d'exécution (d'une tâche) : temps que la tâche doit passer à s'exécuter sur le processeur.

Deadline Monotonic (DM) : algorithme d'ordonnancement en-ligne à priorité fixe pour les tâches. La tâche la plus prioritaire est celle dont l'échéance relative est la plus faible.

$\sim$ $\mathcal{E}$ $\sim$

Échéance absolue (d'une tâche) : date à laquelle la tâche doit avoir fini son exécution

Échéance relative (d'une tâche) : durée de temps dont dispose la tâche pour s'exécuter.

Earliest Deadline First (EDF) : algorithme d'ordonnancement en-ligne à priorité fixe pour les requêtes. La plus grande priorité à la tâche ayant l'échéance absolue la plus proche.

Exécutif temps réel : architecture logicielle comprenant l'ensemble de toutes les primitives nécessaires à l'exécution des tâches ainsi que l'ordonnanceur.

$\sim$ $\mathcal{F}$ $\sim$

Facteur d'utilisation (Analyse) : Calcul, pour une tâche, du ratio composé de la durée d'exécution divisé par la période

$\sim$ $\mathcal{G}$ $\sim$

Gigue sur activation : délai entre la date d'activation d'une tâche et l'instant auquel elle est prête à être exécutée.

$\sim$ $\mathcal{H}$ $\sim$

Hyperpériode : intervalle de temps d'étude d'un ordonnancement.

$\sim$ $\mathcal{L}$ $\sim$

Least Laxity First (LLF) : algorithme d'ordonnancement en-ligne à priorités dynamiques qui à chaque instant donne la plus grande priorité à la tâche ayant la laxité dynamique la plus faible.

$\sim \mathcal{M} \sim$

Machine à traitement par lot parallèle : machine capable d'exécuter plusieurs tâches dans un unique lot en parallèle.

Machine à traitement par lot en série : machine exécutant plusieurs tâches dans un unique lot les unes après les autres.

$\sim \mathcal{O} \sim$

Ordonnancement : partage du temps processeur entre plusieurs tâches.

$\sim \mathcal{P} \sim$

Période (d'une tâche) : durée séparant deux activations successives d'une tâche temps réel.

$\sim \mathcal{R} \sim$

Rate Monotonic (RM) : algorithme d'ordonnancement en-ligne à priorité fixe pour les tâches. il alloue la plus haute priorité à la tâche ayant la plus petite période.

Ratio de compétitivité : c'est la garantie de performance calculée pour un algorithme en-ligne. C'est l'unique valeur en analyse de compétitivité qui permet de comparer les algorithmes en-ligne.

Requête : la tâche à chacune de ses activations génère une requête à exécutée (c'est une copie de la tâche).

~ $\mathcal{S}$ ~

Séquence d'exécution : c'est l'exécution construit par un algorithme en ordonnançant une configuration.

Simulation : technique d'évaluation d'algorithmes en-ligne par l'étude de leur comportement moyen.

Surcharge du processeur : situation d'ordonnancement qui conduit à avoir une durée d'exécution cumulée des tâches plus grande que le temps processeur restant.

Système temps réel : systèmes pour lesquels la correction ne dépend pas seulement des résultats, mais également du temps auquel ils sont fournis.

~ $\mathcal{T}$ ~

Tâche : entité de base des programmes informatiques.

Tâche à suspension : tâche qui pendant son exécution est suspendue par le processeur pour qu'elle lance des opérations externes.

Tâche apériodique : tâche dont l'activation n'est pas connue à l'avance. Généralement ses contraitnes temporelles si elle en a sont souples.

Tâche indépendantes : des tâches sont indépendantes lorsuq'elles n'ont entre elles aucune contrainte de précédence.

Tâche périodique (ou temps réel) : tâche dont l'activation est récurrente ou périodique.

Tâches à départ simultané (ou synchrone) : tâches qui ont toutes la même date d'activation.

Tâches à échéance sur requête : tâches dont l'échéance relative est égale à la période.

Tâche sporadique : tâches à contraintes strictes et dont les activations successives sont séparées par un délai minimum.

Technique de l'augmentation de ressources : extension de l'analyse de compétitivité qui consiste à donner plus de ressources à l'algorithme en-ligne (plus de processeurs ou des processeurs plus puissants) qu'à l'algorithme hors-ligne.

Temps continu : temps qui est dense ou complet.

Temps de réponse (Analyse) : le temps de réponse d'une tâche est la différence de temps entre la fin de son exécution et sa date d'activation.

Temps discret : temps correspondant à une succession d'instants.

Test d'ordonnançabilité : algorithme consistant à tester, au préalable à un ordonnancement, si une configuration est ordonnançable par un algorithme.

Test d'ordonnançabilité approché : test basé sur des estimations plus ou moins fines pour l'ordonnançabilité d'un algorithme. Ces tests sont inexactes mais moins coûteux en calculs qu'un test exact.

Test d'ordonnançabilité exact : test basé sur un calcul exact de l'ordonnançabilité d'un algorithme.

$\sim \mathcal{V} \sim$

Validation : analyse du comportement d'un algorithme en-ligne.

Index

Index

Résumé : Durant cette thèse, deux problèmes d'ordonnancement en-ligne ont été étudiés. Le premier problème concerne l'ordonnancement temps réel de tâches à suspension. Nous avons établi des résultats sur la difficulté à résoudre un tel problème d'ordonnancement (complexité, anomalies d'ordonnancement et non-optimalité des algorithmes en-ligne). Nous avons établi la non-compétitivité d'algorithmes en-ligne pour deux critères de performances même quand ceux-ci disposent de plus de ressources que l'adversaire. Enfin, nous avons étudié avec l'analyse de compétitivité différents tests d'ordonnançabilité. Le second problème se rapporte à l'ordonnancement par une machine à traitement par lot. Plusieurs algorithmes en-ligne compétitifs ont été présentés pour des problèmes dont la taille des lots est non bornée dont αH^{∞} qui fait partie des meilleurs algorithmes en-ligne pour le problème général (son ratio de compétitivité est égal à la borne inférieure du problème, $\frac{1+\sqrt{5}}{2}$).

Contributions to on-line scheduling problems : real time scheduling of self-suspending tasks, scheduling a batch processing machine.

Abstract : In this report, two scheduling problems are studied, using the competitive analysis. The first problem is the scheduling of independant hard real-time tasks with self-suspensions. We first establish results about the difficulties of this scheduling problem (computationnal complexity, scheduling anomalies and non-optimality of on-line algorithms). We have prooved the non-competitiveness of on-line algorithmes for two performance criteria even if they have a faster processor than the adversary. Finaly, we have studied several feasibility tests established for this problem with the competitive analysis. The second problem concerns the single batching machine scheduling problem. We have proposed three on-line competitive algorithms for problems with unbounded batch sizes. We defined the algorithm αH^{∞} that is a best possible on-line algorithm for the general problem because its competitive ratio is equal to the lower bound, $\frac{1+\sqrt{5}}{2}$.

Secteur de Recherche :

Sciences et technologies de l'information et de la communication
Informatique et applications

Mot-clés : Algorithme en-ligne, Analyse compétitivité, Validation, Méthode d'ordonnancement

Laboratoire d'Informatique Scientifique et Industrielle
École Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 - 1, avenue Clément Ader - BP 40109 - 86961 Futuroscope
Tél : 05 49 49 80 63 - Fax : 05 49 49 80 64