Chapitre 1

Ordonnancement temps réel monoprocesseur

1.1. 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 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].

Chapitre rédigé par Pascal RICHARD et Frédéric RIDOUARD.

1.1.1. *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, ...
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 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 on 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.

1.1.1.1. *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. Deux tâches τ_1 et τ_2 sont à départ simultané si leurs dates d'activation sont égales (c.-à-d. $r_1 = r_2$).
Le paramètre d qui dénote la date d'échéance de la tâche. Date à partir de laquelle, la tâche doit avoir terminée 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), 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 appelé *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*.
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 : $U_l = \sum_{i=1}^{n} \frac{C_i}{D_i}$.

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

– Le temps de réponse d'une tâche : R, c'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.

1.1.1.2. *Tâches apériodiques*

Une tâche apériodique a une date d'activation qui n'est connue qu'à l'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 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.

1.1.1.3. *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.

1.1.2. *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 à priori.

1.1.2.1. *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. 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 [AUD 95]* 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.

– 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 pour une configuration pouvant être ordonnancée en respectant toutes les contraintes temporelles des tâches le sera avec l'algorithme A. Et inversement, si A ne peut pas ordonnancer une configuration de ce type alors aucun algorithme ne pourra l'ordonnancer.

1.1.2.2. *Validation et ordonnancement*

La validation et l'ordonnancement d'un système sont très souvent considérés comme des problèmes disjoints, bien qu'ils sont 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.

1.1.2.3. *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 est égale au *ppcm* [1] des périodes des tâches et est désignée sous le nom d'*hyperpériode*. 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 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.
Dans toute la suite de ce chapitre, nous nous limitons volontairement aux systèmes temps réel ordonnancés en-ligne. Pour une comparaison entre l'approche hors-ligne/en-ligne, nous renvoyons à l'article [XU 00].

1.2. Principaux algorithmes d'ordonnancement en-ligne

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

---

1. 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 1.1. *Exemple d'affectation de priorités selon* RM

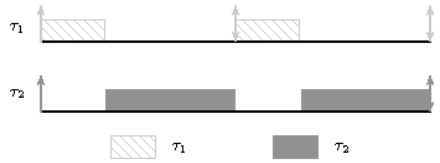

Figure 1.1. *Ordonnancement de la configuration du tableau 1.1 avec l'algorithme* RM

1.2.1. *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 1 *Les priorités selon l'algorithme* RM *sont inversement proportionnelles aux périodes des tâches. La tâche la plus prioritaire est celle ayant la plus petite période.*

Exemple 1 *Exemple d'affectation de priorité et d'ordonnancement d'une configuration I sous* RM *:*
Le tableau 1.1 présente l'affectation de priorité 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 le présente la figure 1.1, 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, indépendantes, à départ simultané au démarrage et à échéance sur requête. Si l'une de ces deux conditions est relâchée,* RM *n'est plus optimal.*

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 1.2. *Exemple d'affectation de priorités selon* DM

1.2.2. *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 2 *Les priorités selon l'algorithme Deadline Monotonic sont inversement proportionnelles aux échéances relatives des tâches. La tâche la plus prioritaire est celle ayant la plus petite échéance relative.*

Exemple 2 *Exemple d'affectation des priorités d'une configuration selon* DM *:*
Le tableau 1.2 montre l'affectation de priorité 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, indépendantes et à départ simultané au démarrage. À noter que si les tâches sont à échéance sur requête,* RM *et* DM *sont équivalents.*

1.2.3. *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 3 *Earliest Deadline First attribue, à chaque instant t, la priorité la plus grande à la requête ayant la plus petite échéance absolue.*

Exemple 3 *Exemple d'ordonnancement d'une configuration de tâches sous* EDF. *La configuration est celle présentée pour* DM *(Tableau 1.2) :*

La longueur de l'ordonnancement à tester est de 24 unités de temps (24 = ppcm(4,6,8)) (cf. Paragraphe 1.1.2.3). 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 1.2). Par conséquent jusqu'à l'instant* 6, *le même ordonnancement sous* EDF *est obtenu que sous* DM. *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].*

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

Least Laxity First [MOK 83] est un algorithme à priorité dynamique. Les priorités assignées par LLF aux requêtes sont inversement proportionnelles à la valeur de la laxité dynamique des requêtes.

Définition 4 *Least Laxity First attribue à chaque instant* t *la priorité la plus élevée à la requête ayant le plus faible laxité dynamique. La laxité de la* k^e *requête de* τ_i *à un instant* t *est égal 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 *Exemple d'ordonnancement d'une configuration de tâches sous* LLF. *La configuration est celle prise pour montrer un exemple d'affectation de priorité par* DM *(Tableau 1.2) :*
Cette configuration 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 1.2). 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].*

1.3. Extension de l'ordonnancement

Dans ce paragraphe, nous présentons plusieurs extensions classiques de l'ordonnancement temps réel.

1.3.1. *Gigue sur activation*

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 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 94] pour les algorithmes à priorité fixe et [SPU 96a] pour les algorithmes à priorité dynamique.

1.3.2. *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 la plus grande priorité à la tâche ayant manquée son échéance. Les tâches suivantes se retrouvent ainsi retardées et manquent 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) permettent d'améliorer l'efficacité des algorithmes d'ordonnancement en-ligne et plus particulièrement EDF [CHA 05, PHI 97]

1.3.3. *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 priorité (*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*) [ABE 98] et *Total Bandwidth Server* (*TBS*) [SPU 94, SPU 95]. Nous renvoyons aux références pour la définition de ces différentes approches.

1.3.4. *Anomalies d'ordonnancement*

Une anomalie d'ordonnancement est liée à l'instabilité de l'ordonnanceur. Ce problème, bien connu en environnement multiprocesseur [GRA 69], se formule de la façon suivante : **un système ordonnançable avec une charge de travail maximum peut être non ordonnançable lorsque la charge processeur diminue (p. ex. réduction des durées des tâches, augmentation des périodes des tâches, augmentation de la vitesse du cpu...**.
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 quelques exemples de système monoprocesseur présentant des anomalies d'ordonnancement : les systèmes non-préemptif, 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).

1.3.5. *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.

1.3.6. *Partage de ressources*

Lorsqu'un ensemble de tâche 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 critique et empêche ainsi une tâche de plus haute priorité de s'exécuter et l'*interblocage* qui survient lorsque deux tâches ayant besoin de deux mêmes ressources critiques pour leur exécution en possède chacun 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].

1.4. 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**.

1.4.1. *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 ordonnnancer n'excède pas les capacités du processeur.

Définition 5 *L'analyse du facteur d'utilisation est un test d'ordonnançabilité qui revient 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)**. 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 6 *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 7 *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).*

1.4.2. *Définition des principaux concepts*

Nous présentons maintenant les concepts nécessaires à l'étude de ces techniques d'analyse de l'ordonnançabilité d'un système de tâches.

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

Définition 8 *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.-à-d. 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. 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 pour 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 9 *(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 (level-i busy period).*

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

Définition 10 *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 sera 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 sera alors approché puisque la demande processeur aura été surestimée. Ceci introduit donc du **pessimisme** dans le test d'ordonnançabilité, c.-à-d. qu'une borne supérieure de la demande processeur est calculée. Le test sera uniquement une condition suffisante d'ordonnançabilité : si le test renvoie vrai alors le système est ordonnançable, sinon on ne peut pas conclure.

Exemple 5 *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.*

• *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.*

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.

Exemple 6 *Voici un exemple introduisant une simplification en jouant sur la périodicité des activations des tâches : Ordonnancement EDF de tâches sporadiques : le pire temps de réponse d'une tâche τ_i survient dans une periode 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.*

1.4.2.2. *Evaluation 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 par 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 :

$$W(t) = \sum_{j=1}^{n} rbf(\tau_j, t) \qquad [1.1]$$

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 sytèmes 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 $[0, t[$:

$$W_i(t) = \sum_{j=1}^{i} rbf(\tau_j, t) \qquad [1.2]$$

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

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 :

$$r_i + (k-1)T_i + D_i \leq t \quad \Rightarrow \quad k \leq \frac{t - r_i - D_i}{T_i} + 1$$

$$r_i + kT_i + D_i > t \quad \Rightarrow \quad k > \frac{t - r_i - D_i}{T_i}$$

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 t dans l'intervalle $[0, t[$:

$$dbf(0,t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - r_i - D_i}{T_i} \right\rfloor + 1\right) \times C_i \qquad [1.3]$$

Exemple 7 *Considérons un système de trois tâches à départ simultané dont les paramètres sont indiqués dans le tableau 1.3. 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 ordonnaçable (alors qu'elle le serait avec une échéance égale à la période). La figure 1.2 présente les fonctions $W_3(t) = rbf(t)$ et $dbf(t)$.*

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, comme 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 vigilent afin d'assurer que les valeurs calculées par

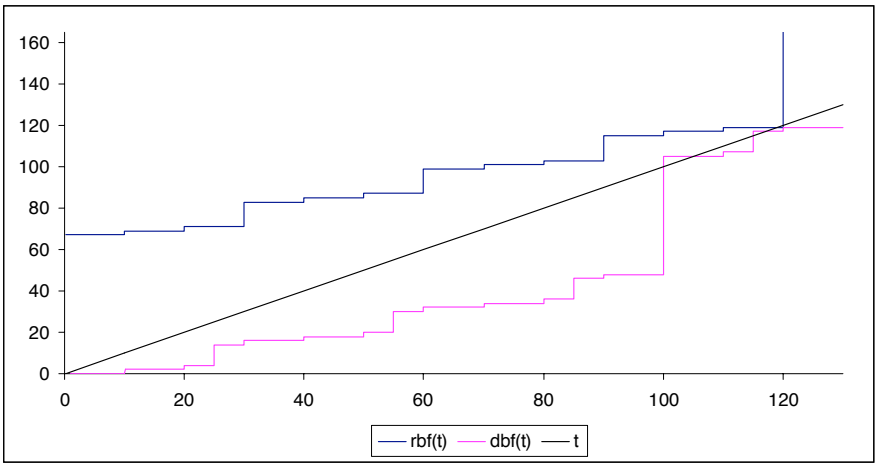

Figure 1.2. *Fonctions $rbf(t)$, $dbf(t)$ et $f(t) = t$ pour le système de tâches du tableau 1.3*

les fonctions rbf ou dbf sont des bornes supérieures de la demande processeur, car 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.

1.4.2.3. *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. Mais des tests approchés reposant sur l'analyse de la demande processeur ont été proposés, comme nous le verrons plus loin.

1.5. Analyse du facteur d'utilisation

Dans ce paragraphe, les tests d'ordonnançabilité basés sur les facteurs d'utilisation sont présentés. Ces tests concernent les algorithmes décrits dans le chapitre 1.2. Ces tests sont exacts ou approchés.

1.5.1. *Tests exacts*

Le test d'ordonnnaçabilité 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) :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \qquad [1.4]$$

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%.

1.5.2. *Tests approchés*

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

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \qquad [1.5]$$

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

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

Le test LIU et LAYLAND en 1973 [LIU 73] (Formule 1.5) 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) :

$$U_\ell = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq 1 \qquad [1.6]$$

Si enfin, l'échéance de toutes les tâches est supérieure à la période, le test 1.4 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.

1.6. Analyse du Temps de réponse

1.6.1. *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.

1.6.1.1. *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 $W_i(t)$ a été définie dans la formule 1.2.

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

$$\begin{aligned} R_i^0 &= \sum_{j=1}^{i} C_j \\ R_i^{n+1} &= \sum_{j=1}^{i} rbf(\tau_j, R_i^n) \end{aligned} \qquad [1.7]$$

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 : $O(n \sum_{i=1}^{n} C_i)$. 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 8 *Sur le système de tâches présenté dans le tableau 1.3, les temps de réponse sont calculés en 3 itérations maximum et les temps de réponse correspondant sont indiqués dans la colonne R_i de ce même tableau.*

1.6.1.2. *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 ordonnancées par EDF.

1.6.2. *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.

1.7. Analyse de la demande processeur

1.7.1. *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.

1.7.1.1. *Ordonnancement à priorité fixe*

Nous présentons dans la suite 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 1.1 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 Scheduled Point Test) :

$$S_i = \{bT_j | j = 1 \ldots i, b = 1 \ldots \lfloor D_i/T_j \rfloor\} \qquad [1.8]$$

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 :

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

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 1.4. *Exécution du test associé au théroème 2 pour analyser le système de tâches du tableau 1.3.*

Exemple 9 *Nous illustrons ce test sur l'analyse de la tâche τ_3 du système de tâches présenté dans le tableau 1.3. Les ensembles de tests sont :*

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

Les calculs de $W_3(t)/t$ pour l'ensemble S_3 sont présentés dans le tableau 1.4. Le système est non ordonnançable puisqu'aucune date $t \in S_3$ ne conduit à vérifier la condition : $W_3(t)/t \leq 1$.

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.

1.7.1.2. *Ordonnancement* EDF

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

Lemme 3 *(Scénario)*

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

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

$$dbf(0, t) = \sum_{i=1}^{n} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i \qquad [1.9]$$

$$= \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i \qquad [1.10]$$

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 relachant la contrainte d'intégralité de l'équation 1.10, 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 10 *Nous illustrons le fonctionnement de ce test sur le système de tâches du tableau 1.3. Le facteur d'utilisation du processeur est $U = 0,99$ et ceci conduit à la valeur $t_{lim} = 2380$ (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 1.2 : 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 $O(n \max_{i=1..n}(T_i - D_i))$, 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.

1.7.1.3. *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'ordonnançabilté. Ceci passe notamment par la définition plus générale de la fonction de la demande cumulée du processeur.

Définition 11 *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 :*

$$\forall t_1 < t_2 \qquad dbf(t_1, t_2) \leq t_2 - t_1 \qquad [1.11]$$

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 1.11 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 1.11 ?

1.7.2. *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 ordonnaçable**, 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 $\overline{rbf}(\tau_i, t)$:

$$\overline{rbf}(\tau_i, t) = rbf(\tau_i t) \quad \text{Si} \quad t \leq (k-1)T_i \qquad [1.12]$$

$$= C_i + t\frac{C_i}{T_i} \quad \text{Sinon} \qquad [1.13]$$

La demande processeur approchée, est alors définie par :

$$\overline{W}_i(t) = C_i + \sum_{j=1}^{i-1} \overline{rbf}(\tau_i, t) \qquad [1.14]$$

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 $1/\epsilon$:

$$\overline{S}_i = \{bT_j | j = 1 \ldots i-1, b = 1 \ldots k\} \qquad [1.15]$$

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 FPTAS. 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.

1.8. Conclusion

La complexité des problèmes d'ordonnançabilité n'est pas abordée dans ce papier. Pourtant, 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 1.3.4) pour le problème considéré ou que le problème d'ordonnançabilité est **difficile**. 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.
Les travaux sur l'ordonnancement monoprocesseur sont nombreux et ainsi beaucoup de résultats sont connus. Ces résultats peuvent être utilisés pour valider de nombreux systèmes industriels. Mais, dans beaucoup de cas 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].

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.

[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.

[AUD 95] AUDSLEY N., BURNS A., DAVIS R., TINDELL K., WELLINGS A., « Fixed Priority Pre-emptive Scheduling : An Historical Perspective », *Real-Time Systems, Kluwer*, vol. 8, n°2/3, p. 173-198, mars / mai 1995.

[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 (ed). 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 , nov 2004.

[BUR 94] BURNS A., « Preemptive Priority-based scheduling : an appropriate engineering approach », *in : Advances in Real-Time Systems (ed Sang H. Son), Prentice Hall*, p. 225-248, 1994.

[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., *Information Processing Letters*, vol. 73, n°5-6, p. 157-161, 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 : Extact 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. W. S., *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.

[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.

[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., NATALE M. D., 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.

[XU 00] XU J., PARNAS D., « Priority Scheduling versus Pre-run-time Scheduling », *Journal of Real-Time Systems*, 2000.

Index