ORDONNANCEMENT À MIGRATIONS RESTREINTES

5.1 Introduction

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

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

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

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

5.2 Terminologie

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

TABLE 5.1 – Notations employées dans le chapitre 5.

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

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

5.3 Algorithme d'ordonnancement

5.3.1 Travaux existants

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

5.3.2 Laxité

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

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

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

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

seur π_j est donnée par :

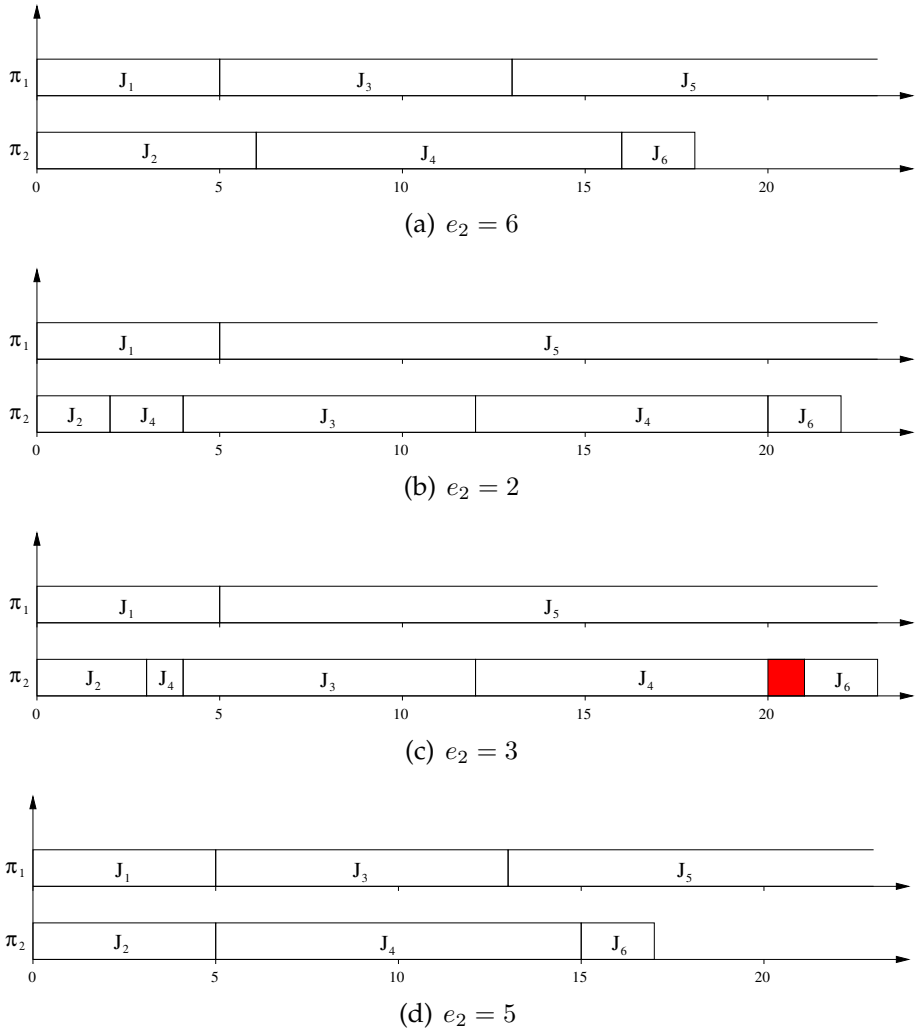
$$L_{i,k}(t) = D_i - C_i - \sum_{J_j \in hp(\pi_j, J_{i,k})} e_j^*(t) \tag{5.1}$$

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

5.3.3 Anomalies d'ordonnancement

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

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



(a) $e_2 = 6$

(b) $e_2 = 2$

(c) $e_2 = 3$

(d) $e_2 = 5$

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

algorithme n'est donc pas prédictible.

5.3.4 Description de l'algorithme

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Viabilité

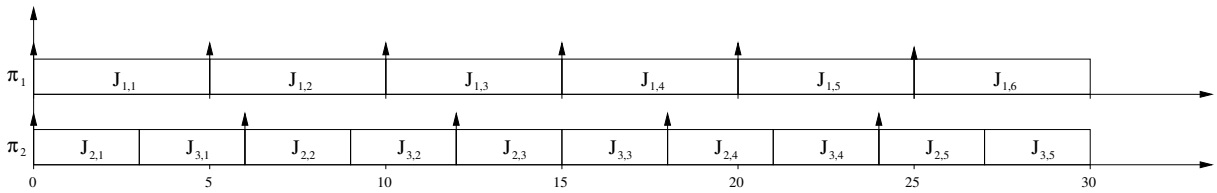
La viabilité d'un algorithme d'ordonnancement, ou tout du moins la viabilité du test d'ordonnançabilité qui lui est associé, est une propriété importante. Elle apporte notamment la garantie que l'algorithme puisse être implanté sans que des anomalies d'ordonnancement ne se produisent. Nous montrons dans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

5.4.1 Résistance aux diminutions de coûts d'exécution

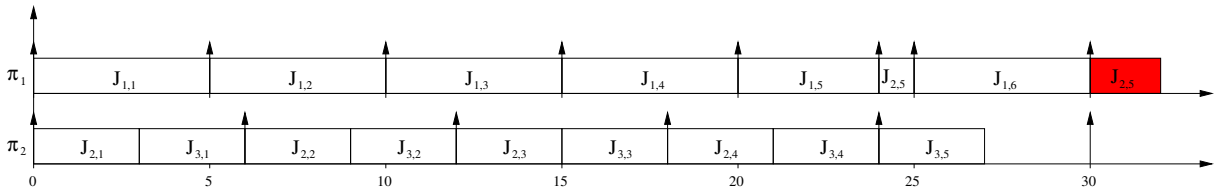
L'anomalie qui se produit avec l'algorithme (FTP-FJП) de Ha et Liu est imputable à la définition de la disponibilité processeur qui lui est associée. En effet, avec cet algorithme, un processeur est disponible pour une instance J_i si il est soit inactif, soit en train d'exécuter une instance de priorité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

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

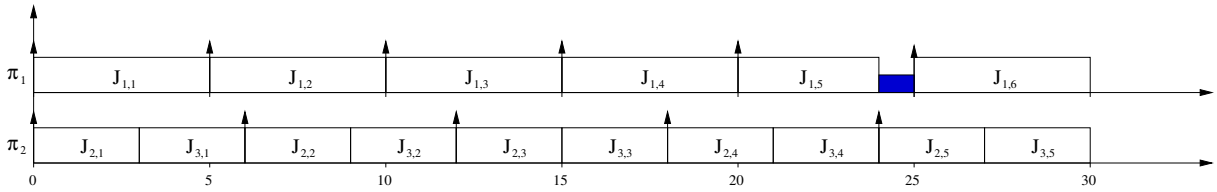
Dans la figure 5.2(a), nous représentons un ensemble d'instances ordonnançables. L'ensemble de tâches qui les engendre est caractérisé par $\{\tau_1(O_1 = 0, C_1 = 5, D_1 = 5, T_1 = 5), \tau_2(O_2 = 0, C_2 = 3, D_2 = 6, T_2 = 6), \tau_3(O_3 = 0, C_3 = 3, D_3 = 6, T_3 = 6)\}$.



(a) Ensemble d'instances ordonnançable.



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



(c) Ensemble d'instances ordonnançables avec oisiveté.

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

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

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

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

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

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

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

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

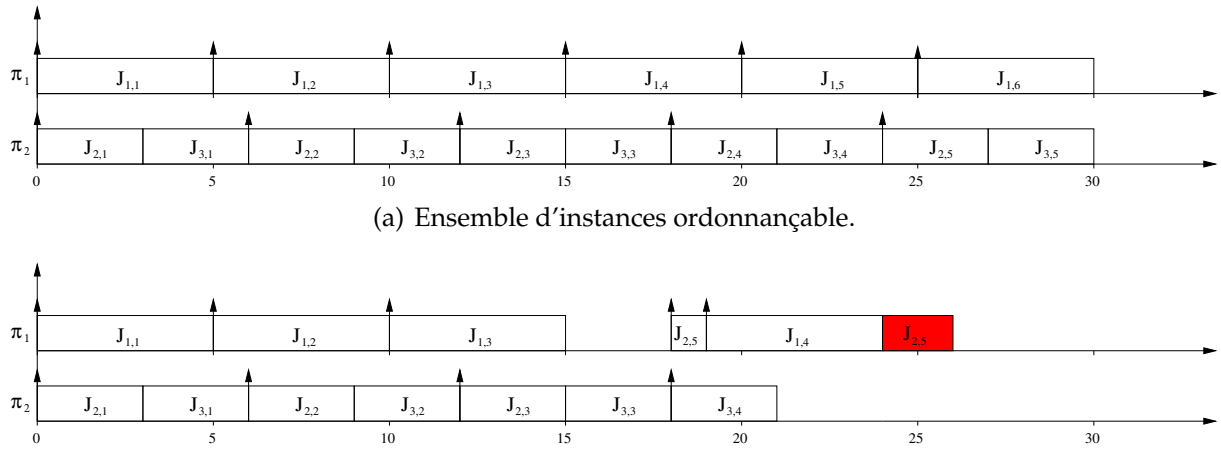
5.4.2 Résistance aux augmentations d'échéances

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

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

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

5.4.3 Résistance aux augmentations de périodes



(a) Ensemble d'instances ordonnançable.

(b) Ensemble d'instances non ordonnançable.

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

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

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

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

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

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

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

5.5 Tests d'ordonnançabilité

5.5.1 Test nécessaire et suffisant

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

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

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

– $S_1 = O_1$;

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

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

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

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

L'intervalle commence à l'instant X_1 correspondant à l'instant où ont été activées les instances qui se terminent après S_n. X_1 est défini récursivement par :

– $X_n = S_n$;
– $X_i = O_i + \left\lfloor \frac{X_{i+1}-O_i}{T_i} \right\rfloor T_i$.

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

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

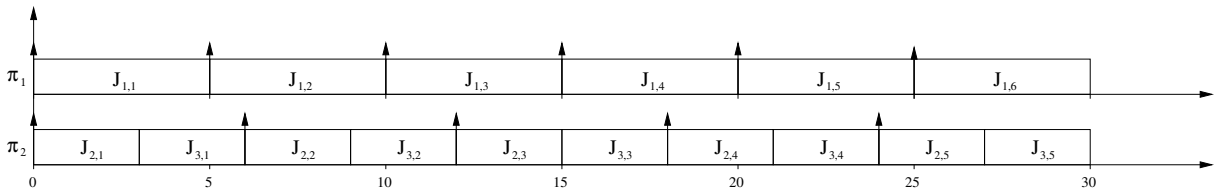
Pour pouvoir décider de l'ordonnançabilité d'un ensemble τ de tâches sur un ensemble Π de processeurs, nous devons identifier le pire scénario d'activations.

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

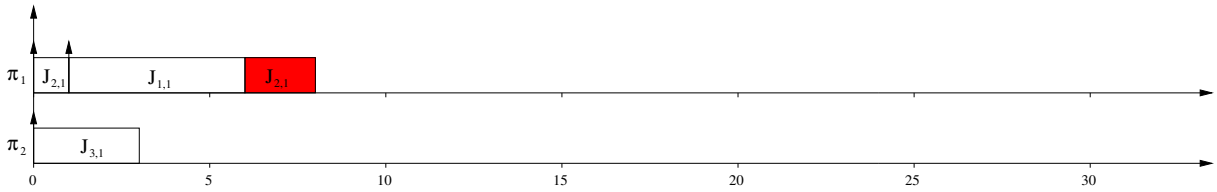
Dans le cas d'un ordonnancement monoprocesseur en priorités fixes, le pire scénario d'activations est le scénario synchrone. Le scénario d'activations synchrone correspond au scénario $O_i = 0$ pour $1 \leq i \leq n$.

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

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



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



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

FIGURE 5.4 – Exemple de scénario synchrone et asynchrone.

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

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

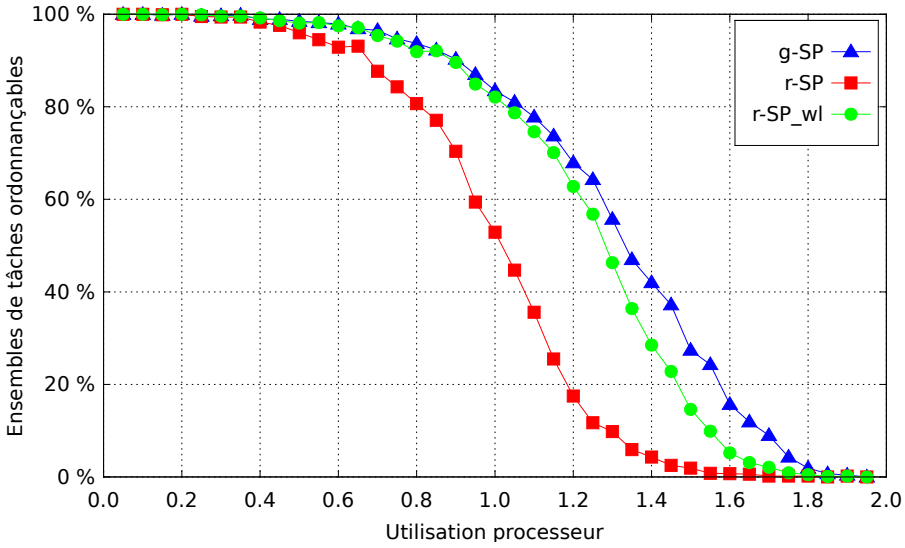


FIGURE 5.5 – Comparaison de l'ordonnançabilité d'algorithmes FTP : algorithme global, algorithme à migrations restreintes et r-SP_wl.

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

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

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

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

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

5.5.2 Test suffisant

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

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

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

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

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

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

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

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

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

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

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

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

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

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

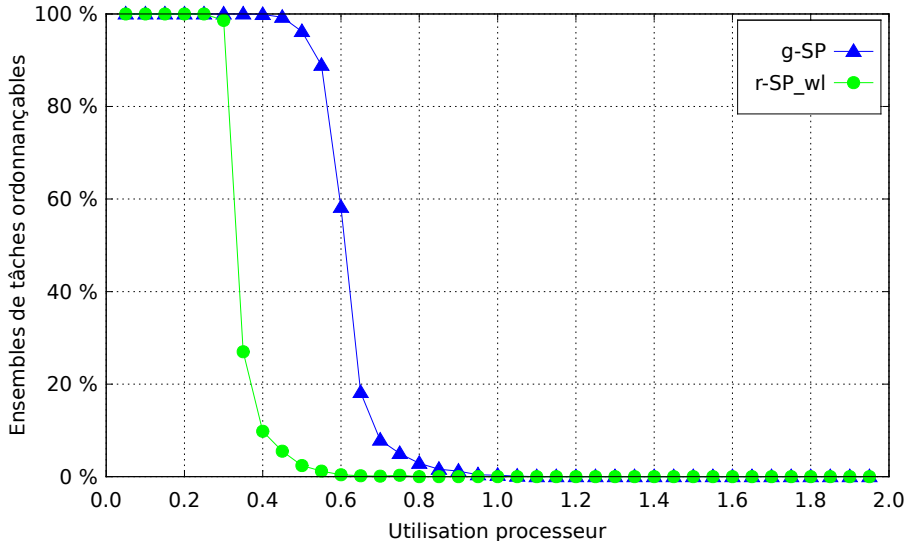


FIGURE 5.6 – Comparaison des bornes basées sur le LOAD de l'algorithme global et de r-SP_wl.

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

5.6 Conclusion

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

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