ALLOCATION DE TÂCHES DÉPENDANTES

4.1 Introduction

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

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

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

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

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

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

4.2 Terminologie et notations

4.2.1 Modèle

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

4.2.2 Notations

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

TABLE 4.1 – Notations employées dans le chapitre 4.

4.3 Protocoles de synchronisation

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

4.3.1 MPCP

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

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

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

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

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

4.3.2 MSRP

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

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

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

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

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

4.3.3 FMLP

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

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

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

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

4.4 Partitionnement de tâches dépendantes

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

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

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

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

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

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

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

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

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

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

avec

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

et

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

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

4.5 Partitionnement robuste

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

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

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

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

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

4.5.1 Recuit simulé

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

4.5.2 Algorithme de partitionnement

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

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

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

Algorithme 4.1: RPSA

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

4.5.3 Initialisation et voisinage aléatoire

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

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

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

4.5.4 Calcul de l'énergie

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

Algorithme 4.2: `calculerEnergie(P)`

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

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

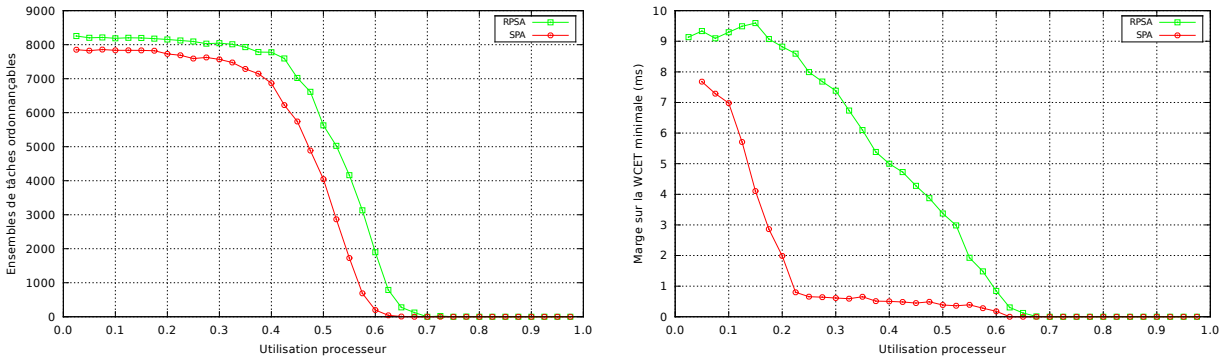
4.6 Évaluation

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

4.6.1 Protocole de simulation

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

4.6.2 Marge sur le WCET



(a) Ordonnançabilité.

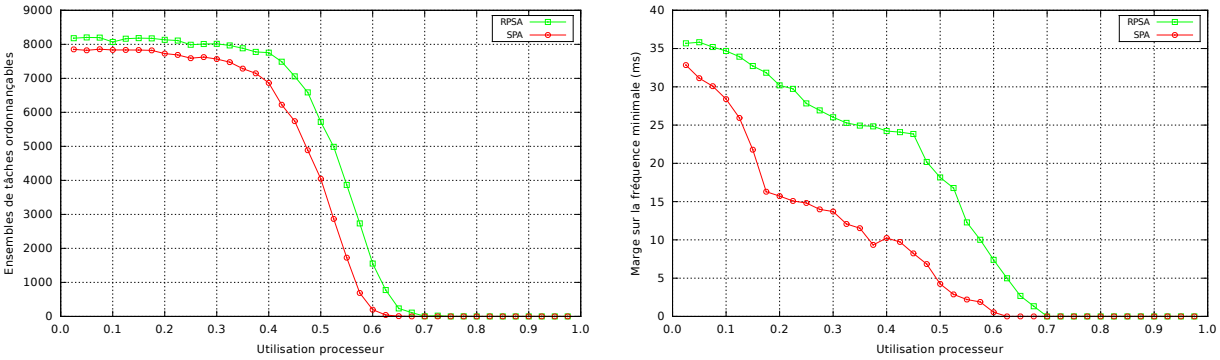
(b) Marge sur la fréquence minimale.

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

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

4.6.3 Marge sur la fréquence

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



(a) Ordonnançabilité.

(b) Marge sur la fréquence minimale.

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

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

4.7 Conclusion

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