Partitionnement temps réel multiprocesseur sous contraintes de qualité de service et d'énergie

Nadine Abdallah

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Nadine ABDALLAH

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
sous le label de l'Université de Nantes Angers Le Mans

Discipline : Automatique et Informatique Appliquée
Laboratoire : Institut de Recherche en Communications et Cybernétique de Nantes (IRCCyN)

Soutenue le 21 fevrier 2014

École doctorale : Sciences et technologies de l'information, et mathématiques
Thèse n° :

Partitionnement temps réel multiprocesseur sous contraintes de qualité de service et d'énergie

JURY

Rapporteurs :	**M^me^ Cécille BELLEUDY**, Professeur, Université de Nice, France **M. Frank SINGHOFF**, Professeur, Université de Brest, France
Examinateurs :	**M. Ye-Qiong SONG**, Professeur, Université de Lorraine, France **M^me^ Claire PAGETTI**, Ingénieur de recherche, ONERA - ENSEEIHT Toulouse, France **M^me^ Audrey QUEUDET**, Maître de conférences, Université de Nantes, France **M. Rafic HAGE CHEHADE**, Professeur, IUT-Saida, Université libanaise, Liban
Directrice de thèse :	**M^me^ Maryline CHETTO**, Professeur, Université de Nantes, France

Tout ce qui ne tue pas rend plus fort
Friedrich Nietzsche

À ma famille, à Najib, à mes amis,
et à la mémoire de ma grand-mère

Remerciements

Je remercie M. Ye-Qiong Song, Professeur à l'Université de Lorraine, qui m'a fait l'honneur de présider le jury de cette soutenance.

Je remercie également M. Frank Singhoff, Professeur à l'Université de Brest et Mme. Cécile Belleudy, Professeur à l'Université de Nice d'avoir bien voulu accepter la charge de rapporteur.

Je remercie Mme. Claire Pagetti, Ingénieur de recherche à l'ONERA de s'être déplacée pour venir apprécier ce travail.

Ma profonde gratitude à ma directrice de thèse, Maryline Chetto, et mes deux encadrants, Audrey Queudet et Rafic Hage Chehade, qui par leur disponibilité, leur confiance, leur soutien, et tous les conseils et retours constructifs qu'ils ont pu me prodiguer, m'ont permis d'acquérir les aptitudes essentielles pour mener à bien mon projet de thèse.

Je tiens à remercier Audrey encore une fois pour toutes ses qualités humaines et pour nos discussions enrichissantes, agréables et motivantes.

Je tiens aussi à mentionner le plaisir que j'ai eu à travailler au laboratoire IRCCyN, et j'en remercie ici tous les membres, permanents et doctorants, qui m'ont accompagnée pendant ce travail de thèse. En particulier, je tiens à remercier Maissa, Jonathan, Sylvain, Sahab, Céline, Adrien, Aleksandra, Inès, Lauriane, Marie, Julien, Adrien, Maxime, Oleg, Oléna, Courtney, Clément, Marija, etc. qui m'ont aidée et supportée durant ma thèse et avec qui l'espace pause a toujours été un moment agréable.

Je remercie également le directeur de l'IRCCyN, M. Michel Malabre, le personnel administratif et tous les membres de l'équipe STR de m'avoir accueillie durant cette thèse.

Je remercie Denis Creusot et Richard Randriatoamanana pour m'avoir sauvé la vie à plusieurs reprises lorque mon PC m'avait lachée.

Enfin, le meilleur pour la fin, je remercie de tout cœur ma famille au Liban tout comme en France, mon amour Najib, et mes amis Donia et Jalal qui durant ces trois ans m'ont toujours soutenue et encouragée.

Table des matières

Table des figures

Introduction générale

Aujourd'hui, les systèmes embarqués prennent une place de plus en plus importante dans notre vie. Dans ces dispositifs embarqués, la gestion de l'énergie électrique est une problématique cruciale. Par exemple, pour un téléphone cellulaire dont la batterie stocke une énergie en quantité limitée et peut être rechargée périodiquement par son utilisateur, cette problématique revient à minimiser l'énergie consommée pour maximiser sa durée d'autonomie. Elle est en général traitée par des méthodes de type DVS (*Dynamic Voltage Scaling*) jouant sur la vitesse du processeur, ce qui affecte directement la consommation énergétique du système. Cependant, cela suppose que les processeurs utilisés dans ce genre de plateforme supportent diverses fréquences de fonctionnement. Par ailleurs, bon nombre de systèmes embarqués de nouvelle génération limitent voire interdisent les interventions humaines, notamment parce qu'ils sont difficilement accessibles de par l'environnement dans lequel ils évoluent (ex : application de surveillance d'une zone forestière ou autoroutière), ou bien parce qu'ils sont déployés en très grand nombre (ex : réseau de capteurs sans fil dense pour un relevé topographique précis). Ces systèmes fonctionnent alors de plus en plus souvent grâce à des batteries ou/et des supercondensateurs qui se rechargent continûment grâce à une source d'énergie renouvelable telle que l'énergie solaire. Concevoir de tels systèmes embarqués, entièrement autonomes, nécessite cependant la résolution d'un certain nombre de problèmes liés à la récolte de l'énergie ambiante, à son stockage et à son utilisation, de façon à assurer une autonomie durable (d'une à une dizaine d'années) et ce, tout en maintenant un respect des contraintes temporelles du système de traitement.

Parallèlement, on observe qu'à mesure que les systèmes temps réel embarqués deviennent de plus en plus complexes, le support d'exécution repose sur des architectures multiprocesseur de par le bénéfice qu'elles apportent par la grande capacité de calcul qu'elles offrent. Nous distinguons trois types de plateformes multiprocesseur selon qu'elles sont constituées de : (i) processeurs identiques où les processeurs ont une même capacité de calcul s, (ii) processeurs uniformes où chaque processeur π_j est caractérisé par sa propre capacité de calcul s_j, (iii) processeurs indépendants où la capacité de calcul $s_{i,j}$ dépend non seulement du processeur π_j mais également de la tâche τ_i qui s'exécute. Les approches d'ordonnancement proposées pour ce type de plateforme sont soit globales, soit par partitionnement.
Dans le cas de l'approche globale, il y a une seule file d'exécution et les tâches sont autorisées à migrer entre les processeurs, ce qui peut engendrer des coûts de migration et de préemption importants. Les approches partitionnées ont, quant à elles, l'avantage d'affecter les tâches de manière unique aux différents processeurs, limitant ainsi les coûts de préemption et n'induisant aucun coût de migration. De plus, dans ce cas, le problème d'ordonnancement multiprocesseur se réduit à résoudre un ensemble de problèmes monoprocesseur.

Par ailleurs, un système est plus fiable s'il peut faire face à des cas de surcharge temporaire de traitement. Parmi les approches de résolution de surcharge, un ensemble de techniques dites tolérantes aux fautes, consistent à écarter (de manière maîtrisée) l'exécution d'un ensemble de jobs afin de réduire la charge processeur, tout en minimisant la dégradation causée par la violation d'une échéance de tâche. Par dégradation, on entendra une Qualité de Service (QoS) moindre, fournie par le système, au niveau de ses résultats. Dans le cas d'un système autonome énergétiquement, la fiabilité passe également par l'assurance que le système ne manquera jamais d'énergie afin d'assurer son traitement. L'anticipation des éventuels cas de famine énergétique

peut, là-encore, être mise en œuvre sur la base de la flexibité offerte par le système au niveau des exécutions des tâches (jobs obligatoires ou optionnels).

Dans le cadre de nos travaux de thèse, nous venons proposer des solutions de partitionnement pour des systèmes temps réel multiprocesseur autonomes énergétiquement autorisant une dégradation contrôlée de la QoS, en particulier dans des phases de famine énergétique et/ou de surcharge de traitement. Nous nous orientons ainsi vers un modèle de tâches temps réel tolérantes aux pertes. Plus précisément, le travail de thèse présenté dans ce rapport propose des solutions au problème de la répartition d'un ensemble de tâches temps réel à contraintes de QoS sur une plateforme multiprocesseur homogène alimentée par une source d'énergie renouvelable. Il convient donc de considérer conjointement deux types de contraintes : temporelles et énergétiques.

Le chapitre 1 constitue une introduction à l'ordonnancement dans les systèmes temps réel, en rappelant les concepts de base liés à cette problématique. Nous résumons par une synthèse bibliographique : (i) l'ordonnancement temps réel multiprocesseur en soulignant l'intérêt de chacune des différentes stratégies, (ii) l'ordonnancement en présence de surcharge de traitement en présentant les principaux schémas d'ordonnancement et les solutions dédiées aux systèmes surchargés et enfin (iii) l'ordonnancement sous contraintes d'énergie basé soit sur des techniques de variation de la fréquence de fonctionnement du processeur, soit sur des techniques de gestion de la mise sous/hors tension du processeur.

Dans le chapitre 2, nous proposons une stratégie de partitionnement d'un ensemble de tâches temps réel périodiques *à contraintes strictes* sur une plateforme multiprocesseur homogène. La technique proposée, appelée KTS, est basée sur le fractionnement de tâches (*task splitting* en anglais) consistant à diviser une tâche n'ayant pas pu être assignée en un ensemble de nouvelles tâches plus légères. Les performances de cette méthode sont comparées à celles d'une autre méthode existante de *task splitting* démontrée efficace mais dont l'inconvénient est d'introduire des contraintes de précédence entre les tâches divisées et donc par là-même d'induire des coûts de migration de contexte d'exécution.

Dans le chapitre 3, nous considérons un modèle de tâches temps réel fermes *sous contraintes de QoS* tolérant occasionnellement l'abandon d'un de leurs jobs selon le modèle Skip-Over. Nous étendons à ce modèle de tâches fermes : (i) les heuristiques de partitionnement classiques et (ii) l'approche KTS proposée dans le chapitre 2. L'idée est d'exploiter la flexibité apportée par le modèle de tâches fermes pour améliorer le taux d'ordonnançabilité (c'est-à-dire, pour un panel d'ensembles de tâches, le ratio d'ensembles de tâches jugés ordonnançables). Ces différentes stratégies sont validées au travers de différentes simulations où nous explorons notamment l'effet du facteur de pertes des tâches sur celles-ci et les comparons entre elles.

Dans le chapitre 4, nous nous intéressons au problème relatif au partitionnement d'un ensemble de tâches temps réel périodiques *à contraintes strictes* sur une plateforme multiprocesseur homogène autonome du point de vue énergétique. Le but est de répartir les tâches de manière à ce que toutes les contraintes temporelles soient satisfaites sur chaque processeur sans qu'il n'y ait à aucun moment une famine énergétique empêchant l'exécution de certains jobs. Earliest Deadline First (EDF) est utilisé comme politique d'ordonnancement. Nous présentons une extension des heuristiques de partitionnement classiques pour le système considéré. Ensuite, nous explorons la façon dont à la fois les critères de tri des tâches temps réel et les contraintes énergétiques peuvent favoriser la performance des différentes heuristiques adaptées. Nous estimons également la capacité minimale des différentes batteries permettant d'obtenir le meilleur taux de réussite et ce, quels que soient le niveau de contrainte énergétique du système et/ou l'heuristique considérée. Nous proposons ensuite l'extension de l'approche KTS aux systèmes temps réel autonomes énergétiquement.

Le chapitre 5 considère la prise en compte conjointe de contraintes temporelles et énergétiques pour un modèle de tâches temps réel périodiques fermes *sous contraintes de QoS*. Nous formalisons le problème du partitionnement et les conditions d'ordonnançabilité associées. L'enjeu

est alors de répartir les tâches en respectant à la fois leurs contraintes temporelles et leurs contraintes énergétiques, tout en assurant l'exécution des jobs obligatoires permettant de garantir la qualité de service minimale requise pour le système.

En conclusion, nous résumons les principales contributions et présentons les axes de développements futurs.

Chapitre 1

État de l'art

Ce premier chapitre constitue une introduction aux systèmes informatiques temps réel soumis à différents types de contraintes telles que des contraintes de qualité de service ou des contraintes d'énergie. Nous présentons tout d'abord les concepts principaux des systèmes temps réel et introduisons la problématique relative à l'ordonnancement temps réel. Puis, nous nous focalisons sur trois points : (i) les systèmes temps réel multiprocesseur, (ii) les systèmes temps réel tolérants aux fautes et (iii) les systèmes temps réel soumis à des contraintes d'énergie.

1 Le temps réel : vocabulaire et modèles

1.1 Le concept de temps réel

L'acception du concept de *temps réel* est très large et bien loin de ce que l'on peut imaginer. Un système temps réel n'est pas un système "qui va vite" mais un système capable de réagir à des stimuli externes dans des délais spécifiés (contraintes temporelles). Nous nous concentrons donc sur la définition du temps en tant que donnée physique mesurable. Une application temps réel est donc un ensemble d'activités auxquelles sont associées des contraintes temporelles. La définition du temps réel largement adoptée dans le domaine est celle de Stankovic [Sta88] :

Définition 1.1 *La correction d'un système temps réel dépend non seulement du résultat logique des calculs mais aussi du temps auquel les résultats sont produits.*

En informatique, on parle d'un système temps réel lorsque ce système est capable de contrôler un procédé physique à une vitesse adaptée à l'évolution du procédé contrôlé. Les systèmes informatiques temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat délivré. Autrement dit, les différents enchaînements possibles des traitements du système garantissent que chacun de ceux-ci ne dépassent pas leurs limites temporelles ; on parle alors d'échéances temporelles des traitements.

1.2 Les spécificités des systèmes temps réel

1.2.1 Définitions et caractéristiques majeures

La définition d'un système temps réel largement adoptée est la suivante [Que06] :

Définition 1.2 *Nous qualifions de système temps réel tout système dont le fonctionnement est assujetti à l'évolution dynamique dans le temps d'un procédé extérieur avec lequel il intéragit et dont il doit contrôler le comportement en exploitant des ressources souvent limitées.*

Le comportement que doit adopter un système temps réel répond à différentes caractéristiques. Nous citons six caractéristiques majeures qui le définissent :

- L'exactitude *logique* et *temporelle* : le système doit être en mesure de fournir des sorties qui concordent avec les entrées du système et ceci dans le respect des contraintes temporelles.
- La *prévisibilité* : la raison d'être du système est de garantir que toutes les tâches (dans toutes leurs configurations d'exécution) satisferont leurs échéances. Celles-ci doivent donc être prévues et exécutées dans les contraintes de temps spécifiées. Pour garantir cela, nous nous plaçons toujours dans le *pire-cas*.
- Le *déterminisme* : le système temps réel réagit toujours de la même façon à un événement entrant, c'est-à-dire que le système produit le même événement sortant.
- La *fiabilité* : le système répond à des contraintes de disponibilité. Les composants matériels et logiciels du système doivent être fiables, c'est-à-dire qu'ils doivent être capables de délivrer un traitement correct des informations reçues à des intervalles de temps bien définis.
- La *tolérance aux fautes* (*fault-tolerant* en anglais) : afin de respecter la contrainte de fiabilité, certains systèmes temps réel sont conçus de façon à être tolérants à certaines fautes qui risquent de survenir et ainsi provoquer l'arrêt du système.
- La *complexité* : chaque système temps réel possède une certaine complexité qui provient non seulement du *non-déterminisme* de l'environnement extérieur avec lequel il interagit (les évènements du monde externe se produisent souvent de manière asynchrone et apparaissent dans un ordre imprévisible) mais aussi des fonctionnalités qu'il réalise.

1.2.2 Taxinomie des systèmes temps réel

Les systèmes temps réel sont classés selon le niveau de criticité de leurs contraintes temporelles [Liu00]. On parle alors de systèmes :

- **temps réel dur** ou *à contraintes strictes* (*hard* en anglais) pour lesquels le non respect d'une contrainte temporelle entraîne la faute du système pouvant engendrer des conséquences catastrophiques (en termes de vies humaines, d'impact sur l'environnement, voire sur l'économie) sur le système lui-même ou son environnement. C'est-à-dire qu'en condition nominale de fonctionnement du système, tous les traitements du système doivent impérativement respecter toutes leurs contraintes temporelles ; on parle alors de systèmes temps réel strict ou dur. On peut citer parmi ces systèmes le contrôle de trafic aérien, les systèmes de conduite de missiles, la supervision de centrales nucléaires, etc.
- **temps réel souple** ou à *contraintes relatives* (*soft* en anglais). Cette contrainte est moins exigeante quant au respect absolu de toutes les contraintes temporelles. Le dépassement des échéances peut provoquer des dégradations acceptables sans conséquences graves sur le système. On pourra citer comme exemple le décalage entre le son et l'image dans une projection vidéo.
- **temps réel ferme** (*firm* en anglais) qui est une sous-classe du temps réel souple pour laquelle le manquement occasionnel des échéances est autorisé. Ce type de système tolère donc le dépassement des échéances mais à la différence des systèmes à contraintes *soft*, ces dépassements sont quantifiés. Dans ces systèmes, la qualité est quantifiée : la mesure du respect des contraintes temporelles prend la forme d'une donnée probabiliste que l'on appellera *Qualité de Service (QoS)*. Cette dernière est relative à un service particulier tel que le nombre d'images rendues par seconde dans une vidéo sans que la personne qui la visualise ne s'aperçoive des transitions, ou relative au comportement du système dans son ensemble (nombre de traitements qui ont pu être rendus dans les temps, tous services confondus par exemple), ou les deux combinés. Citons par exemple le cas d'une projection vidéo (perte de quelques trames d'images en assurant une qualité minimale de 25 images/s pour que l'utilisateur ne puisse s'apercevoir des transitions entre les différentes trames).

1.2.3 La notion de Qualité de Service (QoS)

La définition de la qualité de service est très vaste ; elle englobe de nombreux domaines d'étude y compris la satisfaction subjective de l'abonné. On rencontre cette notion dans le domaine des systèmes temps réel, des réseaux de communication, des systèmes distribués ou encore des applications multimédia. Toutefois, les aspects de qualité de service qui sont traités dans ces différents domaines se limitent souvent à l'identification de paramètres qui peuvent être observés et mesurés. Cependant, d'autres types de paramètres de QoS qui sont de nature subjective et qui dépendent donc de l'opinion subjective de l'usager existent mais ne seront pas utilisés dans le cadre de nos travaux de thèse.

La qualité d'un service est une notion dépendante du service que l'on étudie. Selon le type de service envisagé, la QoS pourra résider dans le débit (vitesse de téléchargement ou diffusion d'une vidéo), le délai (pour les applications interactives ou une communication téléphonique), la disponibilité (accès à un service partagé) ou encore le taux de paquets perdus (pertes des paquets audio ou vidéo qui ont une influence sur la voix ou la vidéo transmise).
Dans les systèmes temps réel, la qualité de service mesure la capacité du système à respecter les contraintes temporelles. Nous adopterons donc la sémantique suivante :

Définition 1.3 *La qualité de service correspond au taux de jobs (ou instances d'une tâche) ayant été exécutés dans le respect de leur échéance temporelle.*

1.3 Caractérisation et modélisation des tâches temps réel

1.3.1 Définitions

Du point de vue du processeur, une tâche est une activité qui consomme des ressources de la machine informatique (de la mémoire et du temps CPU). Une application temps réel est constituée d'un ensemble de tâches (*tasks* en anglais). Le terme tâche désigne la partie de code informatique résultant de la compilation d'un langage de haut niveau qui sera exécutée par le processeur. Une tâche peut être exécutée une multitude de fois durant la vie du système. Nous pouvons citer comme exemple une tâche qui régulièrement relève la température d'un capteur. Nous appelons alors *travail* ou *job* d'une tâche une exécution ou occurrence de celle-ci. Ainsi, une tâche est constituée d'un ensemble infini de travaux.
Dans un environnement multitâche, à tout instant, chaque tâche peut être dans l'un des états suivants :

- **Élue** ou **en cours** (*running*) : c'est le cas de la tâche en train de s'exécuter. Quand il n'y a qu'un processeur, une seule tâche est en cours d'exécution à un instant donné (celle-ci est choisie selon la politique d'ordonnancement considérée et le mode d'exécution : préemptif ou non),
- **Prête** (*ready*) : c'est le cas d'une tâche en attente de la disponibilité de la ressource de traitement,
- **Suspendue** (*suspended*) : c'est le cas des tâches qui sont en attente d'événements qui provoqueront leur réveil.

La Figure 1.1 illustre les états possibles d'une tâche ainsi que les transitions d'un état à l'autre.

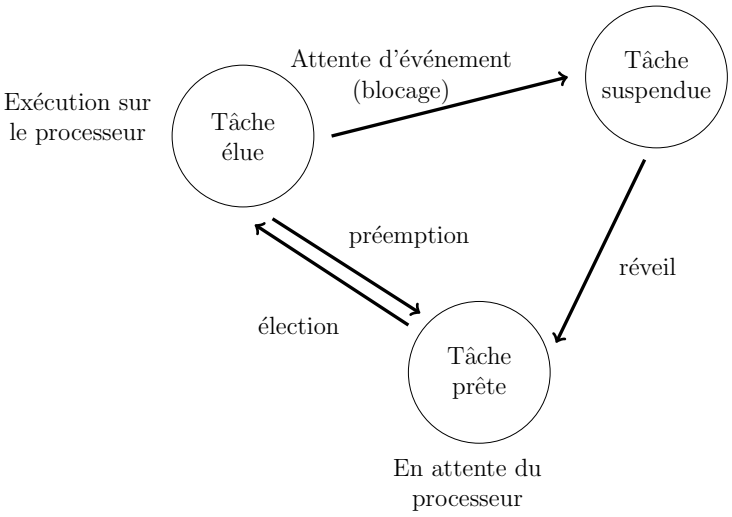


FIGURE 1.1 – États actifs d'une tâche temps réel

En dehors des échéances temporelles associées à chaque tâche, il existe d'autres contraintes dont :

- les contraintes de *précédence* qui définissent un ordre partiel sur l'exécution des tâches. Une tâche n'ayant aucune contrainte de précédence sera qualifiée de tâche *indépendante*.
- les contraintes d'*exécution* qui reposent sur deux modes d'exécution : *préemptif* ou *non-préemptif*. Une tâche préemptible peut être interrompue à tout instant (par une tâche plus prioritaire) et être reprise ultérieurement ou immédiatement sur un autre processeur. Au contraire, une tâche non-préemptible s'exécute à partir du moment où elle est élue et garde l'accès au processeur jusqu'à la fin de son exécution.
- les contraintes de *ressources* qui se traduisent par l'accès aux ressources critiques en exclusion mutuelle pour les tâches qui veulent s'exécuter.
- les contraintes de *placement* qui imposent à une tâche à s'exécuter sur un ou plusieurs processeurs donnés.

Avant de définir les modèles périodiques et sporadiques de tâches, il convient d'introduire un modèle plus général duquel ces derniers dérivent. Ce modèle de base n'est autre que le modèle des jobs (ou travaux).

1.3.2 Modélisation des jobs

Un job (ou travail) est caractérisé par trois paramètres, tel que précisé dans la définition ci-dessous :

Définition 1.4 *Un job est caractérisé par le triplet* (a, e, d) *où :*

- a *est l'instant d'arrivée (*release time *en anglais),*
- e *est la durée d'exécution (*computation time *en anglais),*
- d *est l'échéance absolue (*absolute deadline *en anglais).*

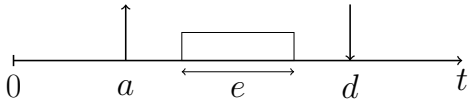


FIGURE 1.2 – Modèle d'un job

Autrement dit, un job qui arrive à l'instant a nécessite e unités de temps d'exécution qui doivent lui être attribuées dans l'intervalle $[a, d]$ pour respecter son échéance temporelle (voir Figure 1.2).

Une politique d'ordonnancement des jobs doit être mise en œuvre. Celle-ci est en charge de sélectionner les différents jobs à exécuter sur le ou les processeurs du système. Seuls les jobs actifs peuvent être ordonnancés, à savoir :

Définition 1.5 *Un job est actif à l'instant t lorsque :*
- *le job est arrivé à un instant antérieur à t ($a \leq t$),*
- *son échéance se situe après l'instant t ($t < d$),*
- *le job n'a pas fini son exécution (moins de e unités de temps ont déjà été exécutées).*

1.3.3 Modélisation des tâches temps réel

Comme énoncé précédemment, l'exécution d'une tâche donne donc lieu à un ensemble de jobs. Il existe principalement trois types de tâches, selon la manière dont les jobs sont activés :
- Les tâches périodiques sont activées régulièrement selon une période fixe ;
- Les tâches sporadiques sont activées de manière irrégulière mais avec toutefois une durée minimale entre l'arrivée de deux jobs consécutifs ;
- Les tâches apériodiques sont activées de manière irrégulière sans aucune propriété qui puisse lier les jobs entre eux.

Le cas des tâches apériodiques ne sera pas traité.

Modélisation des tâches périodiques. Une tâche périodique τ_i (O_i, C_i, T_i, D_i) est définie par :
- O_i, l'instant d'arrivée du premier job de la tâche τ_i (aussi appelé *offset* de τ_i du fait que le premier job peut se réveiller à un instant décalé par rapport au temps initial $t = 0$) ;
- C_i, la durée d'exécution pire-cas ou $WCET$ (*Worst Case Execution Time*) de chaque job de la tâche τ_i ;
- T_i, la période de la tâche τ_i notant la durée qui sépare l'arrivée de deux jobs successifs de τ_i ;
- D_i, l'échéance relative de la tâche τ_i aussi appelé délai critique ;

La date d'arrivée ou de réveil $r_{i,j}$ du j-ième job de la tâche τ_i correspond à la date d'exécution au plus tôt de ce dernier. $d_{i,j}$ représente l'échéance absolue ou date critique du j-ième job de la tâche τ_i qui correspond à la date de fin d'execution de ce dernier au plus tard ; un job arrivant à l'instant $r_{i,j}$ a donc son échéance absolue telle que $d_{i,j} = r_{i,j} + D_i$. Le facteur d'utilisation ou charge processeur u_i de la tâche τ_i correspond au taux d'activité du processeur dédié à l'exécution des jobs successifs de la tâche : $u_i = \frac{C_i}{T_i}$;

L'ensemble des paramètres est illustré sur la Figure 1.3.

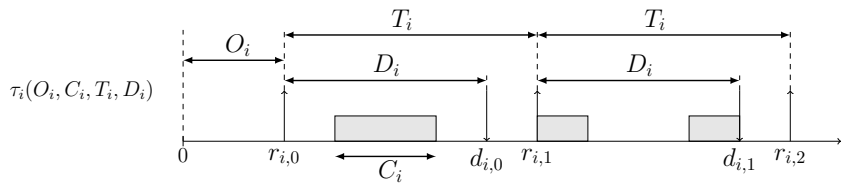


FIGURE 1.3 – Modèle d'une tâche périodique τ_i

Modélisation des tâches sporadiques. Une tâche sporadique τ_i est caractérisée par le triplet (C_i, T_i , D_i) où :
- C_i, représente le temps d'exécution pire-cas de chaque job de la tâche τ_i ;
- T_i, la durée minimale qui sépare deux arrivées successives de jobs de la tâche τ_i.

– D_i, l'échéance relative ou délai critique (un job qui arrive à l'instant t a une échéance absolue à l'instant $t + D_i$) ;

Ce modèle de tâches est illustré sur la Figure 1.4.

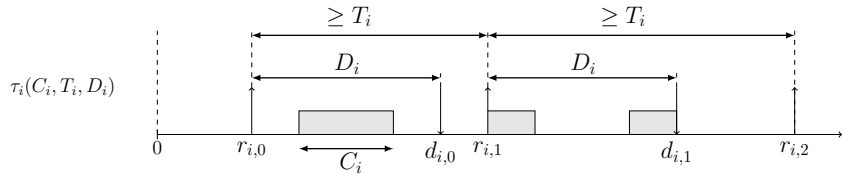


FIGURE 1.4 – Modèle d'une tâche sporadique τ_i

La tâche périodique est un cas particulier du modèle de tâche sporadique. À la différence d'une tâche périodique, les dates de réveil d'une tâche sporadique sont inconnues et sa période T_i correspond à la durée minimale et non à la durée exacte qui sépare deux arrivées successives de jobs d'une même tâche.

Les modèles de tâches se différencient également en fonction du lien présent entre l'échéance D_i et la période T_i.

Définition 1.6 *On dira d'une tâche qu'elle est : (i)* à échéances sur requêtes *si son échéance coïncide avec l'arrivée du job suivant ($D_i = T_i$), (ii)* à échéances contraintes *si son échéance coïncide avant l'arrivée du job suivant $D_i \leq T_i$, (iii)* à échéances arbitraires *s'il n'y a aucun lien entre D_i et T_i.*

Définition 1.7 *On parle de tâches synchrones si celles-ci démarrent au même instant ($\forall i, O_i = 0$), dans le cas contraire, celles-ci sont dites asynchrones.*

2 Problématique de l'ordonnancement temps réel

L'*ordonnancement* est le mécanisme qui permet de choisir la tâche qui va être exécutée par le processeur à un instant donné. L'algorithme chargé d'effectuer ce choix est appelé un *ordonnanceur* (*scheduler* en anglais). Ce dernier doit résoudre les conflits d'accès au processeur suite à des demandes concurrentes d'exécution des différentes tâches. À partir des contraintes imposées (telles que les contraintes de temps, de synchronisation, de ressources), il devra déterminer à tout instant le job à exécuter si celui-ci existe bien, car il est probable qu'aucun travail ne soit prêt à cet instant, le processeur restera alors inactif (*idle* en anglais). Pour cela, l'ordonnanceur peut mettre en œuvre un ou plusieurs *algorithmes d'ordonnancement* qui spécifient une politique d'allocation du processeur aux tâches. On appelle *séquence* la planification temporelle des tâches produite par un algorithme d'ordonnancement pour un ensemble de tâches donné.

2.1 Définitions

Les études d'ordonnançabilité visent à fournir des conditions suffisantes et/ou nécessaires (Définitions 1.13, 1.12 et 1.14) pour déterminer hors-ligne si un ensemble de tâches sera ordonnançable par une politique d'ordonnancement sur une plateforme donnée. Si un mécanisme arrive à ordonnancer toutes les tâches d'un ensemble en respectant leurs contraintes, celui-ci est dit *ordonnançable*. Ci-dessous sont présentées, quelques définitions de termes caractérisant un ordonnanceur :

Définition 1.8 ***Une séquence d'ordonnancement valide*** *établie par un algorithme d'ordonnancement, pour un ensemble de tâches donné, est une séquence pour laquelle toutes les contraintes des différentes tâches de l'ensemble sont respectées.*

Définition 1.9 *(Faisabilité) : Un ensemble de tâches est* ***faisable*** *s'il existe au moins une séquence d'ordonnancement valide dans laquelle toutes les échéances des tâches s'exécutant sur une plateforme donnée sont respectées.*

Définition 1.10 *(Ordonnançabilité) : Un ensemble de tâches est* ***ordonnançable*** *par un algorithme A, si cet algorithme est capable de fournir une séquence d'ordonnancement valide dans laquelle toutes les échéances des tâches considérées sont respectées.*

Définition 1.11 *Un test d'ordonnançabilité détermine, si oui ou non, toutes les contraintes temporelles d'un ensemble de tâches seront respectées lors de l'exécution.*

Définition 1.12 *Dire que A est une condition nécessaire d'ordonnançabilité (resp. de faisabilité) d'un ensemble de tâches B signifie que si A n'est pas vérifiée, il est certain que B n'est pas ordonnançable (resp. faisable). Par contre si A est vérifiée, on ne peut rien conclure quant à l'ordonnançabilité (resp. la faisabilité) de B.*

Définition 1.13 *Dire que A est une condition suffisante d'ordonnançabilité (resp. de faisabilité) d'un ensemble de tâches B signifie que si A est vérifiée, il est certain que B est ordonnançable (resp. faisable). Par contre si A n'est pas vérifiée, on ne peut rien conclure quant à l'ordonnançabilité (resp. la faisabilité) de B.*

Définition 1.14 *Dire que A est une condition nécessaire et suffisante (aussi appelée condition exacte) d'ordonnançabilité (resp. de faisabilité) d'un ensemble de tâche B signifie que si A est vérifiée, il est certain que B est ordonnançable (resp. faisable). Et si A n'est pas vérifiée, B n'est pas ordonnançable (resp. faisable).*

Lorsqu'on a le choix entre plusieurs politiques d'ordonnancement, il peut être utile de pouvoir les comparer. Un premier critère concerne l'optimalité de l'algorithme.

Définition 1.15 *(Optimalité) [GRS96] : Un algorithme d'ordonnancement est dit optimal si pour une classe de tâches, une plateforme et parmi une classe de politiques d'ordonnancement, il est toujours capable de trouver, pour un ensemble de tâches, une séquence valide s'il en existe une.*

De nombreux travaux sur l'ordonnancement vise à déterminer des algorithmes optimaux (Définition 1.15), mais cette optimalité n'est atteinte que dans un cadre précis (hypothèses restrictives sur le type de tâches et/ou sur le type de plateforme). Généralement, les surcoûts liés à l'ordonnanceur (temps de calcul, préemption, etc.) sont négligés. Un deuxième élément de comparaison qui révèle la supériorité d'un algorithme sur un autre est la notion de dominance (Définition 1.16).

Définition 1.16 *Un algorithme A domine un algorithme B si tout ensemble de tâches ordonnançable par B l'est aussi par A et s'il existe au moins un ensemble de tâches ordonnançable par A qui ne l'est pas par B.*

Toutefois, il peut y avoir quelques exceptions pour certains algorithmes qui sont généralement meilleurs que d'autres. Dans ce cas, on ne peut déterminer quel algorithme est dominant. On parle alors d'algorithmes non comparables (Définition 1.17).

Définition 1.17 *Deux algorithmes A et B sont dit incomparables si et seulement si :*

- *il existe un ensemble de tâches qui est ordonnancable par A et non ordonnançable par B,*
- *il existe un ensemble de tâches qui est ordonnançable par B et non ordonnançable par A.*

2.2 Typologie des algorithmes d'ordonnancement

Les algorithmes d'ordonnancement se répartissent selon les caractéristiques liées au système sur lequels ils sont implantés. On aura donc différentes catégories parmi lesquelles :

Monoprocesseur ou multiprocesseur : L'ordonnancement est de type *monoprocesseur* si toutes les tâches ne peuvent s'exécuter que sur un seul et même processeur. Dans le cas contraire, si plusieurs processeurs sont disponibles dans le système, l'ordonnancement est *multiprocesseur*.

Oisif ou non oisif : Un ordonnanceur est dit *non oisif* lorsqu'il possède la propriété suivante : à partir du moment où au moins une tâche est prête et que la ressource processeur est libre, alors l'ordonnanceur élit forcément une tâche et cette dernière commence son exécution sans attendre. L'ordonnanceur fonctionne alors sans insertion de temps creux (*non-idling* ou *work-conservative* en anglais). Dans le cas contraire, si l'ordonnanceur est *oisif*, lorsqu'une tâche est prête, elle peut attendre un certain temps avant d'être élue même si la ressource processeur est libre. On dit que l'ordonnanceur fonctionne par insertion de temps creux (*idling* ou *non work-conservative* en anglais).

Ordonnancement clairvoyant ou non clairvoyant : Nous disons qu'un ordonnanceur est non-clairvoyant lorsque l'ordonnanceur ne sait rien des tâches qu'il doit ordonnancer, en dehors de leurs dates d'arrivées, au moment de leur arrivée. Au contraire, il est clairvoyant lorsqu'il connait *a priori* les caractéristiques des tâches qui arriveront dans le futur.

Préemptif ou non-préemptif : Un ordonnanceur est dit *préemptif* lorsque toutes les tâches qu'il ordonnance sont, elles-aussi, préemptibles. En d'autres termes, l'ordonnanceur peut interrompre l'exécution d'une tâche en cours au profit d'une tâche jugée plus prioritaire. La tâche interrompue reprendra son exécution plus tard dans le temps. Dans le cas d'ordonnanceurs monoprocesseur non-préemptifs, en l'abscence de préemption, il ne peut y avoir d'accès concurrent aux ressources c'est-à-dire qu'à partir du moment où une tâche est élue, celle-ci continue son exécution sans pouvoir être interrompue.

En-ligne ou hors-ligne : Un ordonnancement *hors-ligne* (*off-line* en anglais) signifie que la séquence d'ordonnancement est établie avant le lancement de l'application. Cet ordonnancement convient, en particulier lorsque les tâches à exécuter sont toutes périodiques. Les évènements se reproduisent de façon périodique et l'ordonnancement prend alors la forme d'un plan hors-ligne (ou statique), exécuté de façon répétitive (on parle aussi d'ordonnancement cyclique). Par contre, quand il s'agit d'ordonnancer des tâches parmi lesquelles certaines sont apériodiques, l'absence de cyclicité nous fait nous tourner vers des ordonnanceurs *en-ligne* (*on-line* en anglais). Toutefois, l'ordonnancement *en-ligne* repose souvent sur une analyse hors-ligne afin de garantir le respect des contraintes temporelles à l'exécution dans le pire-cas. Les ordonnanceurs en-ligne permettent par la suite de visualiser la séquence d'ordonnancement des tâches au moment de l'exécution.

Statique/dynamique : Les ordonnanceurs *statiques* fondent leurs décisions d'ordonnancement sur une connaissance complète des paramètres de l'ensemble de tâches ainsi que de leurs contraintes. L'algorithme construit alors une séquence d'ordonnancement qui reste invariable durant toute la durée de vie de l'application associée. À l'inverse, les ordonnanceurs dynamiques fondent leurs décisions sur des paramètres qui peuvent varier (ou de nouvelles occurrences qui peuvent arriver) en cours de fonctionnement du système. Ils prennent en compte l'ordonnancement construit antérieurement pour élaborer de nouvelles décisions.

Centralisé ou distribué : Un ordonnancement est *distribué* si les décisions d'ordonnancement sont prises par un algorithme localement en chaque noeud. Il est *centralisé* lorsque l'algorithme d'ordonnancement pour tout le système, distribué ou non, est déroulé sur un noeud privilégié.

2.3 Critères, métriques de spécification et propriétés de contraintes de validité

Pour le temps réel strict, le critère de validité fondamental est que toutes les tâches respectent toutes leurs contraintes temporelles. Les métriques suivantes nous permettent de vérifier la validité du système :

Le facteur d'utilisation du processeur. Le facteur d'utilisation du processeur pour un ensemble de n tâches périodiques est défini comme la somme des facteurs d'utilisation des différentes tâches s'exécutant sur le processeur :

$$U_p = \sum_{i=1}^{n} u_i = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1.1}$$

Plus précisement, considérons un intervalle de temps débutant à un temps t_d et se terminant à un temps t_f, le facteur d'utilisation d'un processeur dans cet intervalle est calculé comme suit :

$$\sum_{i \epsilon Q} \frac{C_i([t_d, t_f])}{t_f - t_d} \tag{1.2}$$

où Q représente l'ensemble de tâches s'exécutant dans l'intervalle $[t_d, t_f]$ et $C_i([t_d, t_f])$ représente le service reçu par la tâche τ_i sur l'intervalle $[t_d, t_f]$.

Le facteur d'utilisation du système. De manière plus générale, on définit le facteur d'utilisation d'un système :

Définition 1.18 *Le facteur d'utilisation d'un système composé de m processeurs et n tâches périodiques est défini comme suit :*

$$u_{sys} = \frac{1}{m} \sum_{i=1}^{n} u_i = \frac{1}{m} \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{1.3}$$

Les temps creux. Les temps creux d'un processeur (*processor idle times*, en anglais) pour une séquence d'ordonnancement sont les intervalles de temps durant lesquels le processeur est inactif.

Le seuil d'utilisation. Le seuil d'utilisation (*Worst-Case Utilization Bound* ou *Breakdown Utilization*, en anglais) est souvent utilisé pour témoigner de l'efficacité d'un algorithme. Il est défini comme suit :

Définition 1.19 *Le seuil d'utilisation U_{WC}^{A} du processeur (ou du système) pour un algorithme d'ordonnancement A est la valeur du facteur d'utilisation du processeur (ou du système) remplissant les propriétés suivantes :*
- *Tout ensemble de tâches périodiques dont le facteur d'utilisation est inférieur à U_{WC}^{A} est ordonnançable par A.*
- *Pour un facteur d'utilisation supérieur à U_{WC}^{A}, il est toujours possible de trouver un ensemble de tâches périodiques non-ordonnaçable par A.*

Plus ce seuil est grand, plus l'algorithme est jugé performant.

Le facteur de charge du processeur Le facteur de charge d'un processeur désigne la proportion de temps que le processeur dédie à l'exécution des tâches de manière à ce que leurs échéances soient respectées.
Dans le cas de tâches périodiques à échéances contraintes, le facteur de charge du processeur pour un ensemble de n tâches périodiques à échéances contraintes est défini comme étant la somme du rapport des durées d'exécution au pire-cas sur les délais critiques des tâches :

$$\rho = \sum_{i=1}^{n} \frac{C_i}{D_i} \tag{1.4}$$

Dans le cas de tâches périodiques à échéances sur requêtes, le facteur de charge du processeur pour un ensemble de n tâches est équivalent au facteur d'utilisation du processeur et est donné par :

$$\rho = \sum_{i=1}^{n} \frac{C_i}{D_i} = \sum_{i=1}^{n} \frac{C_i}{T_i} (= U_p) \tag{1.5}$$

Les caractéristiques temporelles On peut citer la gigue de démarrage qui représente la mesure de dispersion des délais entre la date de création et la date de démarrage des travaux d'une tâche (*release jitter*, en anglais), le retard, la laxité ou encore le temps de réponse. Le cas particulier de la validation de tâches temps réel strict consiste à garantir que le retard maximal pour ces tâches soit négatif ou nul.

De plus, nous rajoutons d'autres métriques pour témoigner de la validité d'un système temps réel souple ou ferme dont les plus courantes sont :

Le taux de respect. Le taux de respect (*hit ratio*, en anglais) représente la proportion de jobs de tâches respectant leur échéance.

Le taux de garantie. Le taux de garantie (*guarantee ratio*, en anglais) aussi appelé taux d'acceptation (acceptance ratio) représente la proportion de tâches dont l'exécution est garantie par rapport au nombre total de tâches qui demandent à être exécutées.

La valeur dégagée. On associe à l'exécution de chaque tâche une fonction de valeur, et on mesure la somme (ou une autre opération) des valeurs obtenues pendant le déroulement du système [BSS95] (*hit value ratio*, en anglais). On peut citer par exemple pour une tâche temps réel ferme la fonction de valeur suivante : si la tâche se termine avant son échéance, elle dégage la valeur $X > 0$, et 0 sinon.

La cyclicité des séquences d'ordonnancement. Leung et Merill ont fourni un résultat fondamental dans l'ordonnancement préemptif :

Théorème 1 *[LM80] La séquence produite par tout algorithme d'ordonnancement préemptif pour un ensemble de tâches périodiques à départs simultanés est toujours périodique de période H égale au plus petit commun multiple des périodes des tâches de l'ensemble.*

Ce qui implique que pour vérifier l'ordonnançabilité d'un ensemble par un algorithme A, il suffit de vérifier que toutes les échéances dans l'intervalle $[0, H[$ sont respectées pour des tâches synchrones.

2.4 Complexité des algorithmes d'ordonnancement

L'efficacité d'un algorithme d'ordonnancement n'est pas uniquement évaluée en termes de métriques de performance de celui-ci mais aussi en fonction de sa simplicité de mise en œuvre et/ou de calcul. On parle donc de *complexité*. Celle-ci est calculée en évaluant la quantité de ressources en temps (nombre d'instructions) et en espace (mémoire) nécessaire pour la résolution de problèmes au moyen de l'exécution d'un algorithme.

La complexité (temporelle) d'un algorithme correspond au nombre d'opérations élémentaires (affectations, comparaisons, opérations arithmétiques) effectuées par un algorithme et ceci quel que soit le langage de programmation. Ce nombre s'exprime en fonction de la taille n des données. On dit que la complexité de l'algorithme est $O(f(n))$ où f représente le plus grand nombre d'opérations élémentaires que cet algorithme nécessite pour résoudre le problème. Cette fonction prend la forme d'une combinaison de polynômes, logarithmes ou exponentielles. Pour cela, en supposant que n soit la taille du problème, nous cherchons à classer les algorithmes usuels dans les différentes classes de complexité suivantes :

- *Les algorithmes sous-linéaires*, dont la complexité est en général en $O(\log(n))$ et considérés comme très rapides. C'est le cas de la recherche d'un élément dans un ensemble ordonné fini de cardinal n.

- *Les algorithmes linéaires* en complexité $O(n)$ ou en $O(n \log(n))$ sont considérés comme rapides, comme la recherche du minimum dans un tableau composé de n entiers ou les algorithmes optimaux de tri.
- Plus lents, on retrouve *les algorithmes de complexité polynomiale* situés entre $O(n^2)$ et $O(n^3)$. C'est le cas de la multiplication des matrices et du parcours d'un objet en 3D.
- Au delà, les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$ sont considérés comme très lents tout comme les algorithmes exponentiels $O(k^n)$ dont la complexité est supérieure à tout polynôme en n que l'on s'accorde à dire impraticables dès que la taille des données est supérieure à quelques dizaines d'unités.

Lors du développement d'un algorithme, nous visons toujours à atteindre la plus faible complexité possible pour résoudre un problème donné.

P est l'ensemble de problèmes de décision algorithmiques dont la complexité est polynômiale tandis que NP est l'ensemble de problèmes de décision algorithmiques tels que, si une solution possible est donnée, il est possible de vérifier cette solution en un temps polynômial.

Définition 1.20 *Un problème est dit NP-Complet si :*
- *il est dans NP ;*
- *et il est au moins aussi difficile que tout problème de NP.*

Définition 1.21 *Un problème est dit NP-difficile (*NP-Hard *en anglais) s'il est au moins aussi difficile que tout problème de NP.*

2.5 Politiques d'ordonnancement temps réel monoprocesseur

Nous nous plaçons ici dans le cadre simple de l'ordonnancement monoprocesseur non oisif. Toutes les tâches sont considérées indépendantes (pas de contraintes de précédence ni de ressources). Il existe deux types d'assignations de priorités :

1. priorités fixes : aussi appelée assignation à priorité statique, où chaque tâche reçoit une priorité par rapport aux autres à l'initialisation du système et chaque job lancé hérite de la priorité de la tâche auquel il appartient ;
2. priorités dynamiques : la priorité est donnée aux jobs des tâches, et non pas aux tâches. Selon la règle d'assignation utilisée, la priorité évolue au cours du temps tout au long de l'exécution du système.

Nous présentons ci-après les algorithmes d'ordonnancement les plus souvent rencontrés dans la littérature. Nous fournissons pour chaque algorithme une description de son comportement et présentons l'analyse d'ordonnançabilité qui lui est associée.

2.5.1 L'ordonnancement à priorités fixes

2.5.1.1 Rate Monotonic

L'algorithme **Rate Monotonic** (RM) est introduit en 1973 par Liu et Layland [LL73]. Selon cet algorithme, les priorités sont inversement proportionnelles aux périodes des tâches ($T_i < T_j \Rightarrow Prio(\tau_i) > Prio(\tau_j)$). Les conflits lorsque plusieurs tâches possèdent la même période sont résolus de manière arbitraire. En considérant un système composé uniquement de tâches à échéances sur requêtes, l'attribution des priorités en fonction des périodes est la meilleure attribution possible pour avoir une séquence valide. En d'autres termes :

Théorème 2 *[LL73] L'algorithme RM est optimal dans la classe des algorithmes préemptifs à priorités fixes pour les applications constituées de tâches indépendantes à échéances sur requêtes et à départs simultanés.*

Cet algorithme vérifie le théorème de l'*instant critique* : une tâche activée en même temps que toutes les tâches plus prioritaires aura le pire temps de réponse. Liu et Layland ont utilisé cette propriété pour prouver que si les premiers jobs de chaque tâche sont correctement exécutés dans le pire-cas qui est lorsque les tâches sont synchrones alors l'ensemble de tâches est ordonnançable quelles que soient les dates de réveil des tâches. Liu et Layland ont utilisé ce résultat pour développer une condition suffisante pour l'algorithme RM :

Théorème 3 *[LL73] Un ensemble de n tâches indépendantes à échéances sur requêtes est ordonnançable par RM si son facteur d'utilisation U_p vérifie :*

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{1.6}$$

Considérons l'ensemble de tâches $\tau = \{\tau_1(2,2,8,8), \tau_2(1,4,12,12), \tau_3(0,4,24,24)\}$. Nous avons $U_p = \frac{2}{8} + \frac{4}{12} + \frac{4}{24} = 0.75$. D'après le théorème 3, le système est donc ordonnançable. La séquence obtenue est illustrée sur la Figure 1.5 entre les instants $t = 0$ et $t = 26$ qui représente le régime permanent du système ou l'hyperpériode.

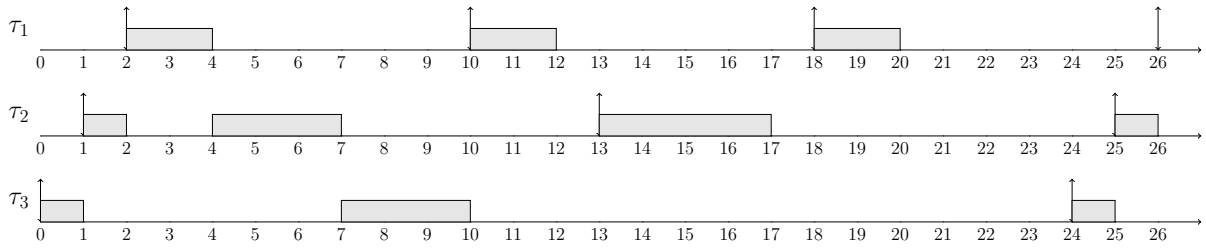


FIGURE 1.5 – Séquence d'ordonnancement selon RM

Cette condition suffisante présente l'avantage d'être simple à évaluer (en $O(n)$). Cependant, cette condition se trouve être pessimiste : lorsque le nombre de tâches n tend vers l'infini, le facteur d'utilisation maximal garantissant l'ordonnançabilité de l'ensemble de tâche tend vers $\ln 2$, soit 69%. Lehoczky *et al.* ont réalisé une étude dans le but de souligner le pessimisme du Théorème 3 en montrant que l'algorithme est capable d'ordonnancer des ensembles de tâches (dont les paramètres sont générés aléatoirement) avec un facteur d'utilisation de 88%.
Dans le but d'améliorer la borne d'ordonnançabilité de l'algorithme RM, un test connu sous le nom de *Hyperbolic Bound (HB)* a été développé par Bini *et al.*. Ce test qui est de même complexité que celui établi par Liu et Layland permet une amélioration du taux d'ordonnançabilité d'un facteur maximum de $\sqrt{2}$:

Théorème 4 *[BBB01] Soit $\tau = \{\tau_1, ..., \tau_n\}$ un ensemble de n tâches périodiques, avec chaque tâche τ_i caracterisée par son facteur d'utilisation u_i. τ est ordonnançable par RM s'il vérifie :*

$$\prod_{i=1}^{n} (u_i + 1) \leq 2 \tag{1.7}$$

D'autres conditions exactes d'ordonnançabilité pour RM ont été dérivées de façon indépendante dans [JP86, LSD89, ABR+93].

2.5.1.2 Deadline Monotonic

Bien que RM soit optimal dans la classe des tâches périodiques à échéances sur requêtes, ce n'est plus le cas lorsque les tâches sont à échéances contraintes ($D_i \leq T_i$). On retrouve donc dans [LW82], un algorithme proposé par Leung et Whitehead connu sous le nom de

Deadline Monotonic (DM) prenant en compte les délais critiques D_i de chaque tâche τ_i tout en gardant le principe d'ordonnancement à priorités fixes. Cet algorithme associe des priorités fixes inversement proportionnelles aux délais critiques ($D_i < D_j \Rightarrow Prio(\tau_i) > Prio(\tau_j)$). Tout comme RM, les conflits sont résolus de manière arbitraire. En considérant un système composé uniquement de tâches à échéances sur requêtes, DM équivaut à l'algorithme RM. Mais dans le cas contraire et en supposant que les tâches sont synchrones, DM se trouve être une stratégie optimale d'ordonnancement :

Théorème 5 *[LW82] L'algorithme DM est optimal dans la classe des algorithmes préemptifs à priorités fixes pour les applications constituées de tâches indépendantes à échéances contraintes et à départs simultanés.*

Leung et Merill ont eux aussi montré que, tout comme RM, DM vérifie le théorème de l'*instant critique*. On retrouve alors la condition suffisante suivante :

Théorème 6 *[LL73] Un ensemble de n tâches périodiques est ordonnançable par DM si :*

$$\sum_{i=1}^{n} \frac{C_i}{D_i} \leq n(2^{\frac{1}{n}} - 1) \tag{1.8}$$

Considérons l'ensemble de tâches $\tau = \{\tau_1(0,2,10,10), \tau_2(0,3,15,8), \tau_3(0,4,30,24)\}$. Nous avons $\frac{2}{10} + \frac{3}{8} + \frac{4}{24} = 0.74$. D'après le théorème 6, le système est donc ordonnançable. La séquence obtenue est illustrée sur la Figure 1.6 entre les instants $t = 0$ et $t = 30$ qui représente l'hyperpériode.

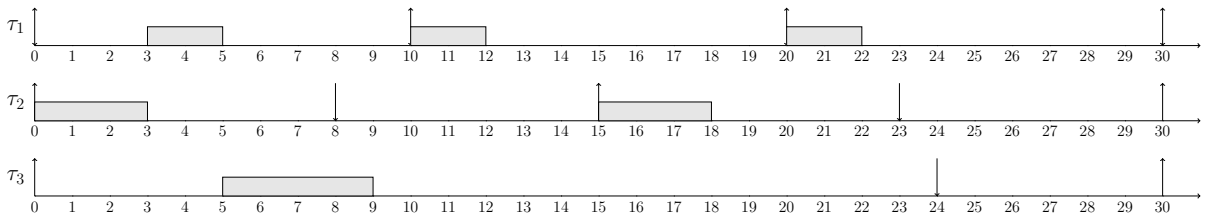


FIGURE 1.6 – Séquence d'ordonnancement selon DM

D'autres études ont bien évidemment été menées [But97, ABRW91] dans le but de trouver une condition exacte d'ordonnancement mais comme dans RM, il s'est avéré qu'elles sont plus coûteuses en termes de complexité.

2.5.2 L'ordonnancement à priorités dynamiques

Les ordonnanceurs à priorités fixes ont, par leur simplicité de mise en œuvre, servi de base dans l'ordonnancement. Les algorithmes à priorités dynamiques sont quant à eux plus difficiles à mettre en œuvre mais offrent souvent de meilleures performances. Les priorités cette fois-ci évoluent en fonction du temps. Nous présentons ci-après deux algorithmes dont l'un est fondé sur l'évaluation des échéances absolues des jobs et l'autre sur la laxité des tâches.

2.5.2.3 Earliest Deadline First

L'algorithme EDF (**Earliest Deadline First**), a été présenté par Jackson en 1955 [Jac55] puis par Liu et Layland [LL73]. Le principe de cet algorithme consiste à élire à chaque instant la tâche dont l'échéance absolue est la plus proche ($d_i < d_j \Rightarrow Prio(\tau_i) > Prio(\tau_j)$). En cas de conflit, la tâche dont la date de réveil est la plus ancienne peut être exécutée la première mais d'autres heuristiques de résolution de conflits peuvent être mises en œuvre. Soit l'ensemble

de tâches $\tau = \{\tau_1(0,1,5,4), \tau_2(0,5,10,10), \tau_3(0,4,20,18)\}$. L'ordonnancement par EDF de cet ensemble de tâches est présenté sur la Figure 1.7.

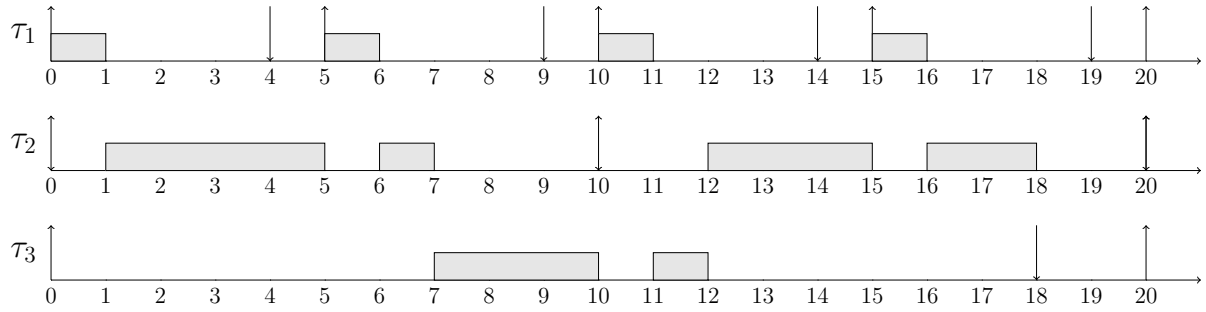


FIGURE 1.7 – Séquence d'ordonnancement selon EDF

Théorème 7 *[Der74] EDF est optimal dans la classe des algorithmes préemptifs pour des tâches périodiques indépendantes.*

En d'autres termes, si un ensemble de tâches ne peut être ordonnancé par EDF (en vérifiant le test du Théorème 8), il ne sera ordonnançable par aucun autre algorithme. Dans le cas non-préemptif, le problème d'ordonnancement est NP-difficile [RRC05]. Par contre, considérant des ordonnanceurs non-oisifs, le problème est de nouveau soluble et comme ont pu le montré Georges *et al.* dans [GMR95], EDF est optimal.

De plus, Liu et Layland présentent dans [LL73] une condition nécessaire et suffisante simple à mettre en œuvre pour des ensembles de tâches à échéances sur requêtes :

Théorème 8 *[LL73, Cof76] Un ensemble de n tâches périodiques à échéances sur requêtes est ordonnançable par EDF si et seulement si son facteur d'utilisation U vérifie :*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{1.9}$$

EDF est certainement l'algorithme le plus populaire. En effet, EDF est un algorithme très performant qui permet d'atteindre un facteur d'utilisation du processeur de 100% sous la condition énoncée dans le Théorème 8. De plus ce résultat est aussi valide pour des ensembles de tâches à départs différés, ce qui illustre bien la supériorité de EDF sur RM ou DM.

Pour des tâches périodiques synchrones à échéances contraintes, nous utiliserons un autre test qui s'avère être lui-aussi nécessaire et suffisant. Ce test intègre un nouveau critère, **la demande du processeur** $h(t)$ (*processor demand criterion* en anglais) proposé par Baruah et al. [BRH90] :

Théorème 9 *Un ensemble de n tâches périodiques à départs simultanés est ordonnançable par EDF si et seulement si :*

$$\forall t > 0, h(t) = \max\left(0, \sum_{i=1}^{n}\left(1 + \lfloor\frac{t - D_i}{T_i}\rfloor\right)\right) C_i \leq t \tag{1.10}$$

La complexité de ce test est pseudo-polynomiale. Cependant, dans le cas où toutes les tâches sont à échéances sur requêtes, les auteurs ont montré que l'analyse d'ordonnançabilité sous EDF se ramenait à une complexité linéaire en $O(n)$.

Aussi, de manière plus générale, des études ont été menées pour déterminer l'ordonnançabilité pour des ensembles de tâches périodiques asynchrones. Ce problème est co-NP-complet [LM80, BRH90]. Leung et Merril [LM80] ont montré qu'il est nécessaire pour cela de vérifier que toutes les échéances sont bien respectées dans l'intervalle $[0, \Phi + 2H]$ ($H = PPCM(T_1, ..., T_n)$ et $\Phi = \max_{i=1..n}\{O_i\}$). Baruah *et al.* ont prouvé le théorème suivant :

Théorème 10 *[BRH90] Un ensemble de tâches τ est faisable sur un processeur si et seulement si :*

1. $U \leq 1$ *et*
2. $\forall 0 \leq t_1 < t_2 \leq \Phi + 2H, h(t_1, t_2) \leq t_2 - t_1$.

$h(t_1, t_2)$ dénote la demande du processeur dans l'intervalle $[t_1, t_2]$ et est donnée par :

$$h(t_1, t_2) = \sum_{i=1}^{n} \eta_i(t_1, t_2) C_i. \tag{1.11}$$

avec $\eta_i(t_1, t_2) = \max\left(0, \lfloor \frac{t_2 - O_i - D_i}{T_i} \rfloor\right) - \max\left(0, \lceil \frac{t_1 - O_i}{T_i} \rceil + 1\right)\Big)$

Le critère de demande processeur est largement utilisé dans l'analyse d'ordonnançabilité multiprocesseur.

Leung and Merill [LM80] ont établi un résultat pour le cas spécifique de l'ordonnancement de tâches asynchrones périodiques sous EDF :

Théorème 11 *[LM80] Considérant un ensemble de tâches $\tau = \{\tau_1, ..., \tau_n\}$, la planification calculée dans l'intervalle $[\max_{i=1..n}\{O_i\} + H, \max_{i=1..n}\{O_i\} + 2H[$ (pas nécessairement le plus court) se répète dans l'intervalle suivant (de même taille) et ainsi de suite, définissant ainsi une cyclité.*

2.5.2.4 Least Laxity First

L'algorithme LLF (**Least Laxity First**), présenté par Mok et Dertouzos [MD78], choisit d'exécuter la tâche dont la date de départ au plus tard (sans violer son échéance) est la plus proche. Pour cela, on définit pour chaque tâche τ_i, une laxité (*slack* en anglais) qui est une fonction du temps :

Définition 1.22 *La laxité d'une tâche τ_i à un instant t se définit comme la durée maximale durant laquelle le processeur peut rester libre à partir de t sans entraîner le manquement de l'échéance de τ_i. En d'autres termes, la laxité d'une tâche τ_i à un instant t est égale à la différence entre l'échéance du job en cours de τ_i et la charge de travail restante, soit :*

$$L(t, \tau_i) = d_i - (t + C_i(t)) \tag{1.12}$$

Nous illustrons cette quantité sur la Figure 1.8. Notons que $C_i(t)$ représente la durée d'exécution restante de la tâche à l'instant t. Pour éviter un manquement d'échéance, nous devons avoir pour chaque tâche τ_i : $\forall t, L(t, \tau_i) \geq 0$.

La priorité maximale est donc attribuée à la tâche possédant la plus faible laxité à la date courante. En cas de conflit, la tâche dont la date de réveil est la plus ancienne peut être exécutée la première (FIFO) mais là-encore, d'autres heuristiques de résolution des conflits peuvent être utilisées.

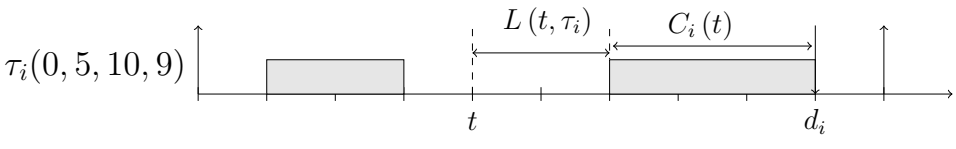


FIGURE 1.8 – Laxité de la tâche τ_i à l'instant t

Soit l'ensemble de tâches $\tau = \{\tau_1(0,2,5,4), \tau_2(0,4,10,10), \tau_3(0,1,20,8)\}$. L'ordonnancement par LLF de cet ensemble de tâches est présenté sur la Figure 1.9.

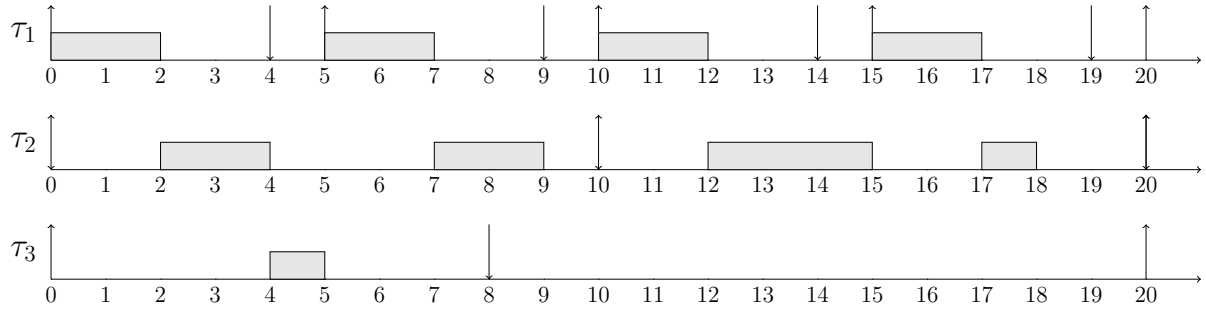


FIGURE 1.9 – Séquence d'ordonnancement selon LLF

L'ouvrage [CDKM00] montre que les conditions d'ordonnançabilité pour l'algorithme LLF sont les mêmes que pour EDF pour un ensemble de tâches à échéances sur requêtes :

Théorème 12 *Un ensemble de n tâches périodiques à échéances sur requêtes est ordonnançable par LLF si et seulement si son facteur d'utilisation U_p vérifie :*

$$U_p = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{1.13}$$

LLF présente les mêmes propriétés d'optimalité que EDF dans le cas d'un ordonnancement préemptif, à savoir :

Théorème 13 *[DM89, Mok83] L'algorithme LLF est optimal pour l'ordonnancement de tâches préemptives, indépendantes et à échéances contraintes.*

Bien qu'il soit aussi performant que EDF, LLF engendre cependant un nombre supérieur de préemptions donc de changements de contexte, ce qui explique qu'il soit aussi peu utilisé dans le cas monoprocesseur [Bim07]. De plus, Georges *et al.* ont prouvé que LLF n'est plus optimal dans le cas d'un ordonnancement non-préemptif [GRS96].

3 Ordonnancement temps réel multiprocesseur

Cette partie s'intéresse à l'ordonnancement de tâches sur des plateformes multiprocesseur. Nous commencerons par souligner l'intérêt d'utiliser de telles plateformes. Les politiques d'ordonnancement présentées par la suite se limitent au cas de processeurs identiques (hypothèse retenue dans le cadre de la thèse). Cependant, certains travaux [CSM97, CM96, HT85, KGJ03, KSMI03, ML04] sont capables de tirer profit d'architectures dites hétérogènes composées de processeurs différents.

3.1 Intêret des plateformes multiprocesseur

Pendant de nombreuses années, les concepteurs de processeurs se sont essentiellement focalisés sur l'augmentation de la fréquence ou l'amélioration des pipelines de ces derniers afin de multiplier la puissance des systèmes informatiques de façon exponentielle et ceci depuis 1960. Mais cette croissance ne peut s'observer infiniment sans que plusieurs problèmes ne surgissent. Tout d'abord, la puissance dynamique P des transistors *CMOS* est proportionnelle aux capacités parasites C, à la fréquence f mais également au carré de la tension V : $P = C.V^2.f$. Dès lors, pour éviter que l'augmentation de la fréquence ne se traduise par une puissance dissipée

accrue, il faudrait réduire la tension d'alimentation tout comme $\sqrt{f}$.
Cependant, certaines limites physiques surgissent :

- la vitesse de propagation des signaux au niveau des interconnexions métalliques et au sein des puces, est bornée par 20 cm/ns ;
- la baisse de la tension réduit la vitesse d'exécution des tâches (ce qui se traduit par une augmentation du temps d'exécution de chaque tâche) ;
- en-deçà d'une certaine taille, la puissance statique des transistors augmente ;
- les pistes d'interconnexion des circuits imprimés sont très sensibles aux parasites et nécessitent une tension différente de celle du processeur pour limiter le bruit ;
- l'échauffement de la puce est également proportionnel au carré de la tension.

Toutes ces limites pratiques des plateformes monoprocesseurs ont donc poussé les chercheurs à se réorienter vers de nouvelles plateformes composées de plusieurs processeurs de façon à continuer à augmenter les performances. En effet, plutôt que d'essayer d'augmenter sans cesse la puissance de calcul d'un processeur, il est préférable de combiner la puissance de plusieurs processeurs et d'améliorer ainsi les performances sans hériter des limites physiques mentionnées ci-dessus.

3.2 Classification des plateformes multiprocesseur

Il est question à présent de décrire les ressources du système qui permettront d'exécuter les applications, à savoir le(s) processeur(s). À l'opposé des systèmes monoprocesseur, les applications disposent de plusieurs processeurs simultanément pour réaliser leurs calculs sur une plateforme multiprocesseur.
La simultanéité (ou parallélisme) d'exécution est cependant limitée par les restrictions suivantes : (i) à un instant t et ceci quel que soit t, un processeur exécute au plus un travail ; (ii) un travail s'exécute sur au plus un processeur à chaque instant.

Nous distinguons trois types de plateformes multiprocesseur selon qu'elles sont constituées de :

- **processeurs identiques.** Les processeurs sont interchangeables et ont une même capacité de calcul s ;
- **processeurs uniformes.** Chaque processeur π_j d'une plateforme uniforme est caractérisé par sa capacité de calcul s_j. Lorsqu'un job s'exécute sur un processeur π_j de capacité de calcul s_j pendant t unités de temps, il réalise $s_j \times t$ unités de travail ;
- **processeurs indépendants.** Pour ce type de plateforme, la capacité de calcul dépend non seulement du processeur mais également de la tâche qui s'execute. Ainsi, on définit une capacité de calcul $s_{i,j}$ associée à chaque couple tâche-processeur (τ_i, π_j) de telle sorte que la tâche τ_i réalise $(s_{i,j} \times t)$ unités de travail lorsqu'elle s'exécute sur le processeur π_j pendant t unités de temps.

Les plateformes identiques correspondent à des plateformes *homogènes* alors que les plateformes uniformes et indépendantes correspondent à des plateformes *hétérogènes*.

Nous nous placerons dans le cadre d'une plateforme à m processeurs ***identiques*** *où* ***m*** *représente le nombre de processeurs qui composent la plateforme.*

3.3 Approches d'ordonnancement multiprocesseur

Cette section présente les différents types de stratégies d'ordonnancement temps réel multiprocesseur existantes. L'ordonnancement pour les systèmes temps réel multiprocesseur poursuit toujours le même objectif à savoir la maximisation de l'utilisation des processeurs tout en garantissant le respect de toutes les échéances. Cependant, nous ne pouvons pas dire que l'ordonnancement multiprocesseur n'est qu'une simple extension de l'ordonnancement monoprocesseur. En effet, l'ordonnancement multiprocesseur est un problème à 2 dimensions : (i)

l'organisation **temporelle** des tâches (stratégie d'ordonnancement) et (ii) l'organisation **spatiale** (allocation des tâches sur les processeurs). Il existe deux types fondamentaux de stratégies multiprocesseur : les *approches globales* et les *approches par partitionnement*.

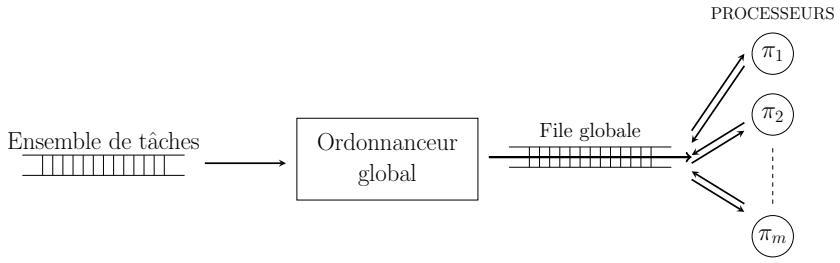


FIGURE 1.10 – Illustration de la stratégie globale

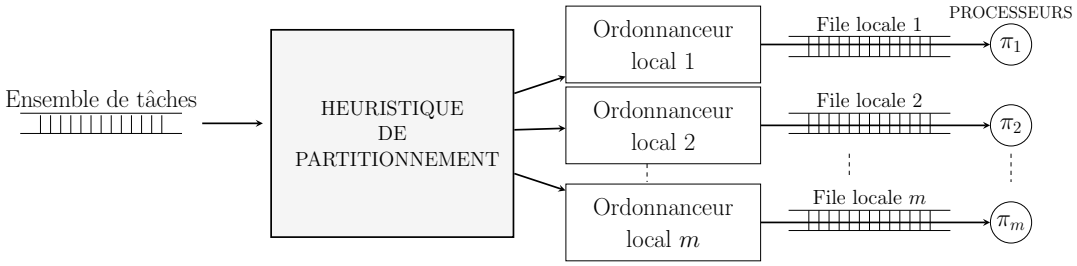


FIGURE 1.11 – Illustration de la stratégie par partitionnement

De ces deux approches pourtant très différentes ont été définies de récentes stratégies intermédiaires appelées *stratégies par semi-partitionnement*.

Avant de définir le comportement adopté lors de l'utilisation de chacune de ces stratégies, nous supposons disposer d'un système composé de n tâches périodiques préemptives à ordonnancer sur une plateforme composée de m processeurs identiques.

Approche globale : l'objectif est d'attribuer, à chaque instant, les m tâches les plus prioritaires aux m processeurs. Pour cela, il existe une seule file d'attente pour l'ensemble des tâches et on utilise une stratégie d'ordonnancement unique qui s'applique sur l'ensemble des processeurs (voir Figure 1.10). Dans ce cas, une propriété essentielle est que la *migration des tâches* soit *autorisée*.

Approche par partitionnement : l'objectif est d'affecter chaque tâche à un processeur ou, autrement dit, de définir m sous-ensembles de tâches attribués à chacun des processeurs. L'objectif réside dans le fait qu'à chaque processeur π_j soit associé un sous-ensemble de tâches Γ_j tel que l'intersection de chaque ensemble soit vide et que l'union des ensembles soit l'ensemble de tâches τ. Notons que les migrations ne sont pas autorisées. De ce fait, le problème multiprocesseur se résume alors à m problèmes monoprocesseurs. Il est ainsi possible d'appliquer des stratégies monoprocesseur sur chacun des processeurs de la plateforme, qui possèdent chacun leur propre file de travaux en attente d'exécution (voir Figure 1.11).

Approche par semi-partitionnement : L'approche semi-partitionnée est dérivée de l'approche partitionnée. Dans cette approche, certains jobs d'une tâche peuvent être exécutés sur des processeurs différents alors que certaines tâches ne sont pas du tout autorisées à migrer, ce qui entraîne des migrations moins nombreuses.

Il est à noter que pour toutes ces stratégies, il existe une condition nécessaire de faisabilité :

Théorème 14 *[Hor74] Un ensemble de n tâches périodiques (ou sporadiques) est faisable seulement si :*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq m \tag{1.14}$$

Nous présentons ci-après les stratégies d'ordonnancement existantes dans le cadre des différentes approches énoncées précédemment.

3.4 Approches globales

Dans cette section, nous présentons les principaux résultats de recherche relatifs à l'ordonnancement global multiprocesseur dans lequel, rappelons-le, les tâches sont autorisées à migrer d'un processeur à l'autre. Il s'agit donc d'appliquer sur l'ensemble des processeurs une stratégie d'ordonnancement globale et donc d'utiliser un seul ordonnanceur pour l'ensemble des processeurs. Si le nombre de tâches n est supérieur ou égal au nombre de processeurs m, à chaque instant, les m tâches les plus prioritaires sont attribuées aux m processeurs, sinon certains processeurs restent inactifs. Une tâche peut commencer son exécution sur un processeur π_j, puis être préemptée par une tâche plus prioritaire, pour reprendre son exécution sur un autre processeur π_k. Ce phénomène est appelé **migration de tâche** et représente une caractéristique fondamentale des approches globales.

L'ordonnancement global présente les avantages suivants par rapport à l'approche par partitionnement :

1. Il y a généralement moins de préemptions et donc moins de changements de contexte, et ceci parce que l'ordonnanceur ne préempte une tâche que lorsqu'il n'y a pas de processeurs inactifs [AJ00].
2. La capacité inutilisée créée sur un processeur lorsqu'une tâche s'exécute sur une durée inférieure à son temps d'exécution pire-cas peut être utilisée par toutes les autres tâches, et non pas seulement celles évoluant sur le même processeur.
3. Si une tâche dépasse son temps d'exécution pire-cas, alors il y a sans doute une plus faible probabilité de défaillance. En effet, le comportement pire-cas de l'ensemble du système correspondant au cas où toutes les tâches s'exécutent selon leur temps d'exécution pire-cas, est moins probable avec plusieurs processeurs qu'avec un seul processeur.
4. L'ordonnancement global est plus approprié pour des systèmes ouverts où il n'est pas nécessaire d'effectuer un équilibrage de charge ou une phase d'allocation des tâches aux processeurs lorsque l'ensemble des tâches change.

L'ordonnancement global présente cependant quelques inconvénients. Dans le cas d'une plateforme monoprocesseur, un ensemble de tâches périodiques ordonnançable reste ordonnançable si ces tâches deviennent sporadiques alors que cette propriété n'est plus valable en multiprocesseur. Contrairement aux algorithmes d'ordonnancement monoprocesseur, les algorithmes d'ordonnancement multiprocesseur présentent des anomalies. En effet certaines modifications des paramètres des tâches qui paraissent intuitivement positives à l'ordonnancement, peuvent parfois rendre un ensemble de tâches non ordonnançable. Par exemple, pour un ensemble de tâches ordonnançable, le fait de diminuer le facteur d'utilisation d'une tâche en augmentant sa période ou en diminuant sa durée d'exécution conduit intuitivement à imaginer conserver un ensemble de tâches ordonnançable, ce qui n'est pas toujours le cas en ordonnancement multiprocesseur.

Phillips *et al.* [PSTW97] ont étudié la manière dont un algorithme en-ligne peut se comporter si l'on augmente la vitesse des processeurs. Ils ont montré qu'un ensemble de tâches qui est ordonnançable sur m processeurs identiques, est ordonnançable avec EDF sur m processeurs identiques mais avec un facteur d'accélération de $(2 - 1/m)$ par rapport à la vitesse initiale. Parmi les algorithmes d'ordonnancement global, les algorithmes PFair sont les plus

connus. Baruah *et al.* ont établi dans [BCPV96] leurs fondements théoriques qui se reposent sur la notion de *proportionate fairness*. Ces algorithmes diffèrent des algorithmes classiques d'ordonnancement dans la mesure où le taux d'exécution d'une tâche est quasi constant. Lorsque l'on considère de grands intervalles de temps, le taux d'exécution d'une tâche sur un intervalle de temps est approximativement égal à son facteur d'utilisation u_i. Cependant, sur de petits intervalles de temps, le taux d'exécution d'une tâche peut varier de manière très importante. Dans les algorithmes PFair, chaque tâche est exécutée à un taux approximativement constant et ce, en divisant la tâche en série de sous-tâches exécutées dans des intervalles identiques appelés *fenêtres*. Dans le cas particulier des architectures multiprocesseur homogènes et des tâches périodiques à échéances sur requêtes, PFair est optimal. Il existe trois variantes de PFair : PF [BCPV96], PD [BGP95] et PD^2 [And00]. La stratégie utilisée par ces algorithmes est proche de celle de EDF dans la façon d'attribuer les priorités aux sous-tâches. À noter que certains algorithmes d'ordonnancement monoprocesseur comme EDF peuvent s'appliquer en ordonnancement multiprocesseur global [DP06], mais ils ne sont pas optimaux et peuvent se révéler très inefficaces. En effet, les travaux fondateurs de Dhall et Liu [DL78] concernant l'ordonnancement global de tâches périodiques à échéances sur requêtes sur m processeurs ont montré que le facteur d'utilisation maximum d'un ensemble de tâches (*maximum utilization bound*, en anglais) représentant le facteur d'utilisation à partir duquel il peut y avoir violation d'échéance pour l'ordonnancement EDF global est $1 + \varepsilon$, pour ε très petit. Cela se produit lorsque l'on cherche à ordonnancer m tâches de courtes périodes/délais critiques ayant un facteur d'utilisation infinitésimale, et une tâche de plus longue période/délai critique ayant un facteur d'utilisation qui se rapproche de 1. Cet *effet de Dhall* a conduit au constat général que les approches globales étaient moins efficaces que les approches par partitionnement. En conséquence, dans les années 1980 et au début des années 1990, la majorité des recherches en ordonnancement temps réel multiprocesseur a reposé sur les approches par partitionnement décrites dans la section suivante.

3.5 Approches par partitionnement

Ces approches sont celles qui ont reçu le plus d'attention dans la littérature, la principale raison étant d'être facilement utilisables pour garantir l'ordonnançabilité à l'exécution. En effet, en utilisant un test d'ordonnançabilité monoprocesseur comme condition d'admission lors de l'ajout d'une nouvelle tâche à un processeur, il est possible de garantir que toutes les tâches respecteront leurs échéances à l'exécution. Rappelons toutefois que cette méthode décompose l'ensemble des tâches en m sous-ensembles $\Gamma_1, \Gamma_2, ..., \Gamma_j, ..., \Gamma_m$, avec Γ_j, le sous-ensemble ordonnancé sur le processeur π_j. De plus, chaque processeur possède sa propre stratégie d'ordonnancement, celle-ci pouvant être différente d'un processeur à un autre. Cela revient à utiliser un ordonnanceur par processeur. Dans cette approche, les tâches ne sont pas autorisées à migrer d'un processeur à un autre [DB11]. Le problème qui consiste à trouver un partitionnement optimal, est équivalent à un problème de remplissage de boîtes de tailles identiques avec des objets de tailles différentes (*Bin-packing problem*, en anglais) [CGMV99] en cherchant à placer le plus grand nombre d'objets possibles dans chaque boîte et en cherchant à minimiser le nombre de boîtes. Les tâches correspondent aux objets et les processeurs aux boîtes. Ce problème est NP-difficile au sens fort [GJ90]. Des algorithmes de complexité polynomiale sont proposés pour le résoudre de façon approchée [BLOS95, OS95, SVC98] en effectuant à la fois le choix des processeurs et la vérification d'une condition d'ordonnançabilité sur chaque processeur (voir Figure 1.12) :

- **Best Fit** (BF), où une tâche est assignée au processeur possédant le moins de capacité inutilisée pouvant l'ordonnancer ;
- **First Fit** (FF), où une tâche est assignée au premier processeur pouvant l'ordonnancer, en partant du premier processeur π_1 ;
- **Worst Fit** (WF), où une tâche est assignée au processeur possédant le plus de capacité inutilisée pouvant l'ordonnancer ;

– **Next Fit** (NF), où une tâche est assignée au premier processeur pouvant l'ordonnancer dans $\pi_j, ..., \pi_m$ (π_j étant le processeur courant). La procédure débute par le processeur π_1.

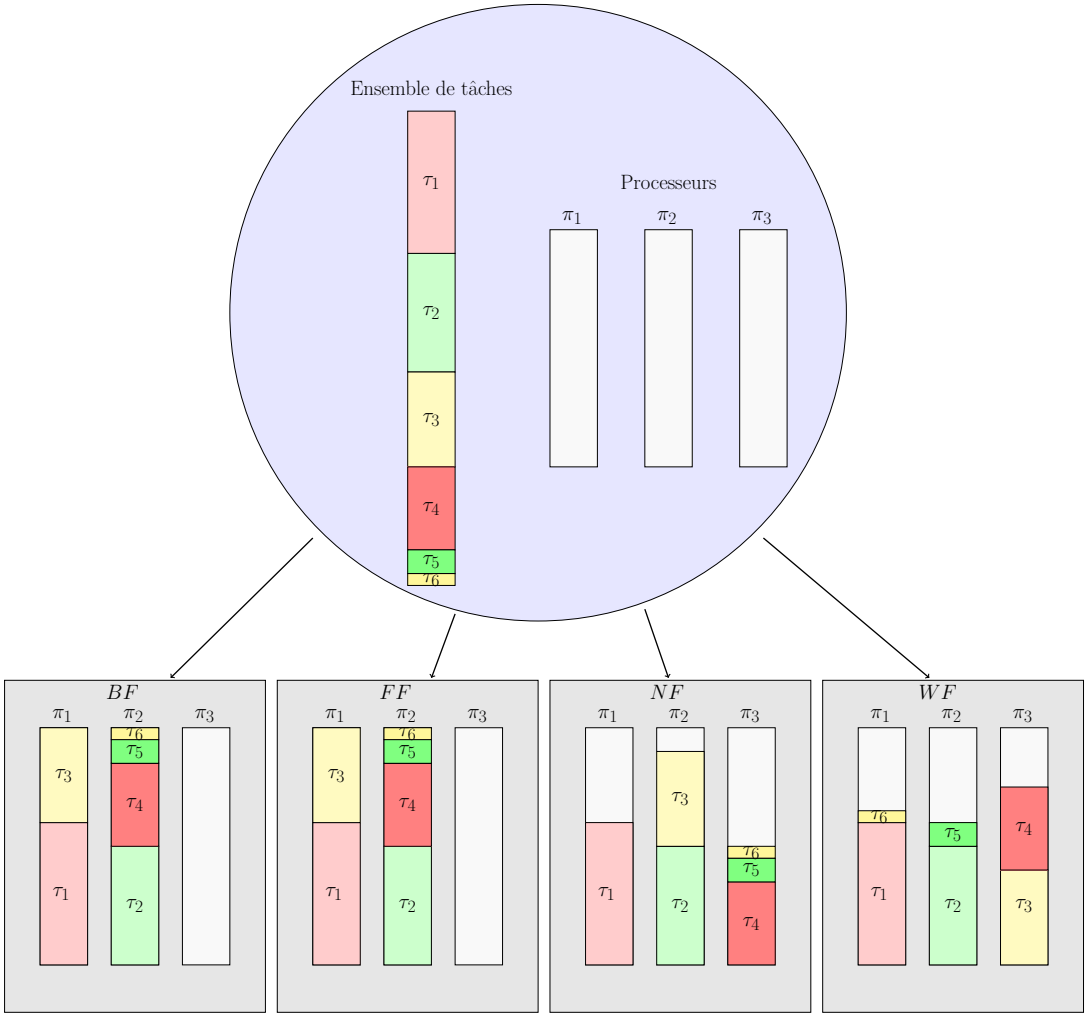


FIGURE 1.12 – Résultats d'assignation selon les heuristiques de partitionnement BF, FF, NF, WF

À noter que certains algorithmes d'ordonnancement monoprocesseur tels que EDF peuvent s'appliquer en ordonnancement multiprocesseur partitionné [LDG01, LGDG00]. De plus, Lopez *et al.* ont montré dans [LGDG00] que tous les algorithmes de partitionnement qui trient leurs tâches dans l'ordre décroissant de leur facteur d'utilisation atteignent de meilleurs seuils d'utilisation des processeurs (comme EDF-BF et EDF-FF). Cependant, les algorithmes de type partitionné souffrent d'un très mauvais seuil d'utilisation du système. Ce seuil est seulement de $\frac{m+1}{2m}$ [ABJ01]. En effet, soit un système de m processeurs avec $m+1$ tâches à ordonnancer. Si chaque tâche a un taux d'utilisation de $0.5+\epsilon$ ($\epsilon > 0$), alors une tâche ne pourra pas être affectée à un processeur. Le seuil d'utilisation du système est de $\frac{(m+1)(0.5+\epsilon)}{m}$.

3.6 Approches hybrides

L'approche hybride ou semi-partitionnée combine les avantages des deux approches énoncées précédemment. Dans cette approche, certains jobs d'une tâche peuvent être exécutés sur différents processeurs, ce qui entraîne des migrations moins nombreuses que dans le cas de l'ordonnan-

cement global. En revanche, certaines tâches ne seront pas autorisées à migrer. Ces approches autorisent donc une migration contrôlée de certaines tâches. Elles peuvent donc être vues comme une amélioration des algorithmes partitionnés en ajoutant plus de flexibilité en s'inspirant des politiques globales.

De récentes recherches [AT06, ABB08, KY07, KY08a, KY08b, GSYY10] décrivent l'intéret des approches hybrides. Ce concept a été introduit par Anderson *et al.* dans [ABD05] par l'algorithme EDF-fm dédié aux systèmes temps réel souples pour lesquels les violations d'échéances des jobs n'entraînent pas la faute du système. EDF-fm est un algorithme d'ordonnancement multiprocesseur basé sur l'ordonnancement EDF qui assure des retards d'exécution bornés des tâches du système lorsque leur utilisation est au plus égale à 1/2. Au plus $m-1$ tâches doivent être capables de migrer, et chaque tâche peut migrer entre deux processeurs tout en respectant les frontières de chaque job de manière à ce que l'exécution d'un job ne se fasse que sur un seul processeur. EDF-fm est composé de deux phases : (i) une phase de partitionnement hors-ligne dans laquelle chaque tâche est assignée à un ou deux processeurs, (ii) un algorithme d'ordonnancement en-ligne qui assure des retards bornés. En 2006, Anderson et Tovar ont introduit un nouvel algorithme EKG (*Earliest deadline first with K processors in a Group*) [AT06]. Cet algorithme diffère de EDF-fm du fait que les tâches migrantes sont exécutées dans certains intervalles de temps, tandis que les tâches fixes (non-migrantes) sont ordonnancées selon EDF. EKG permet de borner le nombre de tâches autorisées à migrer et augmente la localité des tâches dans les mémoires. Ainsi le surplus de temps processeur requis (*overhead*, en anglais) induit par les préemptions de jobs et les migrations est limité. EKG est conçu pour ordonnancer des tâches périodiques. EDF-SS [ABB08] est une extension de EKG pour des ensembles de tâches à échéances arbitraires.

Dans [KY08c], Kato et Yamasaki introduisent EDHS (*Earliest Deadline and Highest-priority Split*). Chaque tâche est assignée à un seul processeur, tant que les processeurs disposent d'une capacité inutilisée suffisante pour accepter l'exécution totale de la tâche. Si par contre une tâche ne peut être assignée à aucun processeur, elle est découpée entre plusieurs processeurs. Il en résulte deux catégories de tâches : d'une part les tâches non-migrantes qui sont affectées de manière unique à un processeur et d'autre part, les tâches qui sont autorisées à migrer.

Les algorithmes Ehd2-SIP (aussi appelé EDDHP) [KY07] et EDDP [KY08a] sont similaires à EKG mais diffèrent légèrement dans leurs heuristiques d'allocation des tâches et dans leurs politiques d'ordonnancement. Selon la valeur du paramètre K de EKG, ces algorithmes permettent de réduire le nombre de préemptions comparé à EKG, mais ils ne sont pas optimaux. Dans l'algorithme EDDHP, la plus haute priorité est donnée aux tâches migrantes, et les autres tâches non-migrantes ont des priorités selon EDF. La politique d'ordonnancement garantit que toutes les tâches s'exécutent en respectant leurs échéances contrairement à EDF-fm. EDDHP présente moins de préemptions qu'EKG et garantit un seuil d'utilisation de 65 % pour les ensembles de tâches à échéances sur requêtes. RMDP est une variante de EDDHP mais basée sur l'ordonnancement RM au lieu de EDF [KY08b]. L'algorithme EDHS [KY08c] change vraiment le procédé d'allocation des tâches. Pour les algorithmes précédents, une tâche ne pouvait migrer que sur deux processeurs, alors que EDHS propose de faire un partitionnement classique puis de répartir les tâches non affectées sur plusieurs processeurs. Cependant, chaque processeur ne peut accueillir qu'une tâche migrante. DM-PM [KY09] est une variante de EDHS qui utilise un ordonnancement à priorité fixe, ce qui a pour avantage de simplifier les calculs. En effet, en découpant les tâches en sous-tâches, on obtient naturellement des tâches plus prioritaires selon DM. SPA2 [GSYY10] est une politique qui se base sur des priorités fixes et qui a pour particularité de partager la même condition suffisante d'ordonnançabilité que RM en monoprocesseur. Comparé à RMDP, DM-PM et PDMS-HPTS-DS [LRL09], le seuil d'utilisation est meilleur. EDF-WM [KYI09] s'inspire des avantages apportés par DM-PM mais en se basant sur EDF. NPS-F [BA09], permet d'atteindre une utilisation qui varie entre 75% et 100% (selon le compromis entre utilisation et préemptions).

Dans [BDWZ12], Burns and al. introduisent un algorithme basé sur l'ordonnancement multiprocesseur ayant la propriété de fractionner une tâche rejetée (*task-splitting scheme*, en anglais). Cet algorithme est appelé *EDF Split (DD)*. Son concept repose sur le fait que pendant la phase d'assignation, chaque processeur reçoit des tâches jusqu'à ce qu'aucune tâche ne puisse être ajoutée au processeur π_j sans qu'elle aboutisse à une violation d'échéance. Cette tâche τ_i non attribuée est alors divisée de telle sorte que la première partie puisse être exécutée sur le processeur π_j avec la capacité restante et la seconde passe sur le processeur suivant. La première partie τ_i' de la tâche découpée possède une contrainte qui est que son échéance est égale au temps d'exécution pire-cas tel que le facteur d'utilisation sur π_j soit égal à 1. La deuxième partie de la tâche est attribuée au processeur suivant π_{j+1} avec le temps d'exécution restant et un décalage afin d'éviter que les deux tâches fractionnées ne s'exécutent simultanément. Son échéance relative est également réduite. L'algorithme se poursuit par l'assignation de nouvelles tâches au processeur π_{j+1} jusqu'à ce que le processeur ne puisse plus recevoir d'autres tâches. Une autre tâche est alors choisie et est fractionnée entre le processeur π_{j+1} et π_{j+2}, etc. Les auteurs montrent que EDF Split (DD) apporte une amélioration par rapport aux approches entièrement partitionnées.

Dans nos travaux, nous avons opté pour le développement de stratégies basées sur l'approche par partitionnement du fait que cette dernière réduit le problème multiprocesseur à m problèmes monoprocesseur. Cette approche rend ainsi possible l'application de stratégies monoprocesseur sur chacun des processeurs de la plateforme possèdant chacun leur propre file de travaux en attente d'exécution.

4 Ordonnancement temps réel et gestion de la surcharge

4.1 Description des cas de surcharge

Idéalement, un système fonctionne toujours de manière conforme aux prévisions faites initialement. En réalité, les systèmes peuvent être soumis subitement à des surcharges temporaires de traitement (*transient overload*, en anglais) lorsque les ressources disponibles (en particulier les ressources processeur) ne suffisent plus pour l'exécution de l'ensemble des tâches de l'application.

Définition 1.23 *On dit d'un système temps réel qu'il est en surcharge pendant une période donnée si sur cette période l'utilisation du processeur dépasse la borne autorisée par l'algorithme d'ordonnancement mis en œuvre.*

On distingue plusieurs causes de surcharge associé au système et à son environnement. La surcharge peut provenir d'une mauvaise conception du système lui-même :

- Les durées d'exécution réelles des tâches peuvent s'avérer supérieures aux durées d'exécution pire-cas qui ont été mal évaluées ;
- Des surcoûts de traitement associés à la plateforme peuvent avoir été sous-estimés ou mésestimés ;
- Le test d'ordonnançabilité peut être erronné.

Par ailleurs, le dysfonctionnement d'un périphérique d'entrée peut retarder certaines tâches conduisant ainsi à la surcharge du système. La surcharge peut aussi provenir d'un flux élevé de tâches apériodiques ou d'un grand nombre de tâches périodiques rapprochées dans le temps.

Il est alors impossible en cas de surcharge de respecter toutes les échéances des tâches. Dans certains cas, le comportement global du système peut être compromis. En particulier, s'il est question de tâches critiques, les conséquences d'une surcharge peuvent être catastrophiques sur l'environnement contrôlé. En surcharge, le comportement de Rate-Monotonic RM est tel que lorsqu'un job d'une tâche manque son échéance, les tâches possèdant de plus longues périodes violent leur échéance à leur tour [Del94]. Pour pallier à ce problème, une solution pratique

consisterait à réduire la période des tâches jugées importantes ce qui ne serait sans doute pas sans conséquences. En effet, cette simple action entraîne une augmentation de l'utilisation processeur de la tâche et donc peut engendrer encore plus de violations d'échéances des tâches de l'ensemble. L'algorithme EDF pourtant plus performant que RM le sera moins en cas de surcharge où son comportement devient instable. Locke a montré que EDF est sujet à l'Effet Domino [Loc86]. En d'autres termes, une tâche qui manque son échéance empêche une autre de s'exécuter qui à son tour manque son échéance. Ceci engendre une avalanche de fautes temporelles dégradant rapidement les performances sur les intervalles de surcharge. Ceci est dû au fait que l'algorithme EDF donne toujours la plus grande priorité à la tâche dont l'échéance est la plus proche. En cas de surcharge, EDF ne fournit aucun type de garanties sur l'identité des tâches qui respecteront leurs contraintes temporelles. Ce comportement qualifié d'indéterministe est particulièrement indésirable lorsque des cas de surcharges dus à des modifications de l'environnement subites par exemple surviennent. Pour faire face aux surcharges, le système doit donc être en mesure d'autoriser certaines dégradations de performances de manière à éviter la faute du système et ainsi le maintenir dans un état sécuritaire.

La conception d'un système temps réel doit donc se faire de manière à éliminer une partie des risques en cas de surcharge. Par exemple, il est possible de filtrer les événements générés par l'environnement, pour qu'ils ne dépassent pas une certaine fréquence d'apparition [Sta94]. Mais cette approche n'est pas forcément compatible avec le système considéré.

En présence de surcharge, la solution pour contrôler le comportement du système consiste à borner ou à diminuer les besoins en ressources. Cela suppose :

- que l'on soit capable de détecter en-ligne le début de la surcharge. Ce problème est NP-complet [BHR93] dans le cas général. Cependant, il existe certains cas pour lesquels il est possible de profiter de bonnes propriétés des algorithmes d'ordonnancement qui permettent de détecter le début de la saturation du processeur [KS93]. Sinon, on fait appel à des approximations qui consistent à mesurer le taux d'occupation du processeur afin de prévoir la surcharge (éventuelle) [MS95, BBL01]. La formalisation puis la mesure de la charge sont elles-mêmes des problèmes en tant que tels, puisqu'il s'agit de tenir compte à la fois des ressources processeur demandées globalement, et des contraintes temporelles qui sont locales à chaque tâche, et font partie de la surcharge.
- que l'on accepte le fait que le système force une tâche à se terminer si nécessaire.

En aval, une fois la surcharge détectée ou anticipée, il existe plusieurs méthodes pour diminuer le besoin en ressource processeur : (i) soit certaines tâches ou jobs de tâches sont volontairement abandonnées ; (ii) soit la structure globale du système est modifiée (augmentation du nombre de processeurs par exemple). Nous nous intéressons maintenant à la première méthode. Nous distinguons dans cette approche, plusieurs classes d'ordonnanceurs :

Les ordonnanceurs au mieux (*Best Effort*). Un ordonnanceur *Best-Effort* accepte systématiquement les tâches qui arrivent sans test d'acceptation et les insère dans la file des tâches prêtes. Ainsi, les tâches sont ordonnancées jusqu'à leur terminaison ou jusqu'à dépassement d'échéance. Comme il n'y a aucune prédiction des cas de surcharge, l'ordonnancement est basé sur les priorités données aux tâches. Ce type d'approche est souvent sensible à l'Effet Domino.

Les ordonnanceurs à garantie (*Guarantee*). Selon un algorithme à garantie, chaque tâche qui arrive subit un test d'acceptation empêchant le système d'être surchargé (test d'ordonnançabilité). Si le test réussit, la tâche est acceptée ; sinon elle est rejetée. Ces ordonnanceurs ont l'avantage par rapport aux ordonnanceurs *Best-Effort* de rejeter toute nouvelle tâche pouvant causer la surcharge du système. Cependant, il ne prennent pas en compte les valeurs associées à l'importance des tâches. Une variante de EDF avec test d'acceptation appelé GED (pour Guaranteed Earliest Deadline) est présenté dans [BS93].

Les ordonnanceurs robustes (*Robust*). Dans un ordonnanceur robuste, l'ordonnancement est contrôlé par deux politiques séparées. Les tâches sont ordonnancées selon leurs éché-

ances et rejetées selon leur importance. En d'autres termes, en cas de refus d'une tâche, l'ordonnanceur tente d'abandonner une ou plusieurs tâches de moindre importance. Les tâches rejetées pourront être reprises plus tard si la charge du système le permet. Les ordonnanceurs robustes sont en général plus performants que ceux basés sur les deux modèles précédents [BS93].

En cas de surcharge, certains jobs sont ainsi ignorés ou abandonnés dans le but de permettre à d'autres jobs jugés plus importants de s'accomplir dans le respect de leur échéance. Ainsi, durant une phase de surcharge, une façon intuitive de mesurer la performance d'un algorithme d'ordonnancement consiste à évaluer la proportion de traitement correctement effectuée par l'ordonnanceur. Plus cette valeur est importante, meilleur est l'algorithme.

Un premier courant [JNC+89] a introduit le concept de *fonction valeur* associant une valeur à chaque tâche en fonction de l'instant auquel son exécution s'achève. Ces approches dites *basées sur la valeur* prennent autant en compte les contraintes temporelles des tâches que le coût infligé au système suite à la violation d'une échéance. On cherche ainsi à ordonnancer en priorité les tâches les plus importantes (l'importance de chaque tâche étant reflétée, à tout instant, par sa fonction valeur). En d'autres termes, dans la plupart de ces algorithmes, le critère de rejet en cas de surcharge, consiste à choisir les tâches de moindre valeur, ce qui peut conduire à une sous-utilisation des ressources et mener à des performances médiocres. De plus, un inconvénient majeur de ces approches réside dans la difficulté du choix de fonctions de valeur appropriées, traduisant au mieux l'importance relative des tâches. Baruah et al. [BKM+91] ont montré qu'aucun algorithme en-ligne ne peut garantir une valeur cumulative (*cumulative value*, en anglais ; il s'agit de la somme des importances des tâches garanties) supérieure à 1/4 de la valeur obtenue par un algorithme clairvoyant. De plus, cette approche suppose que le résultat fourni par une tâche n'a de valeur que si celle-ci s'exécute complètement.

Le second courant, consiste à établir des modèles intégrant directement le fait que des tâches puissent être totalement ou en partie abandonnées. Celui-ci regroupe :

- *Les approches à tâches bipartites* pour lesquelles seule une partie de la tâche est garantie hors-ligne (l'exécution du reste dépend de la charge du système). Dans ces modèles le résultat d'une tâche possède toujours une certaine valeur, même si un résultat est dégradé (Mécanisme à échéance [CHB79]) ou approximatif (Calcul Imprécis [LNL87, LLN87]). En cas de surcharge, les algorithmes d'ordonnancement reposant sur ces modèles, garantissent un niveau de QoS minimal, correspondant à l'exécution des parties obligatoires des tâches.
- *les approches à pertes contraintes* pour lesquelles la spécification du modèle précise quels jobs d'une tâche peuvent être ignorés. Cette approche de résolution de surcharge se focalise sur le relâchement des garanties strictes des contraintes temporelles des tâches. Dans un environnement temps réel à contraintes fermes, une violation occasionnelle d'échéance n'a pas de conséquences catastrophiques, et se traduit uniquement par une dégradation de la QoS. Ainsi, la gestion de surcharge dans cette approche repose sur l'abandon contrôlé de certains jobs de tâches pour diminuer l'utilisation effective des ressources et minimiser la dégradation due aux jobs abandonnés. Les algorithmes se basant sur cette méthode devront donc maintenir un nombre d'échéances non respectées inférieur à une valeur limite donnée, sous peine d'entraîner une défaillance au niveau du système.

Nous nous intéressons dans la suite aux approches à pertes contraintes qui constituent l'un des piliers abordés dans les travaux de thèse présentés dans les chapitres suivants.

4.2 Modèles de résolution de surcharge à pertes contraintes

4.2.1 Le modèle (m, k)-firm

La notion de contrainte (m, k)-firm a été introduite par Hamdaoui et Ramanathan [HR95]. L'ordonnancement à garantie (m, k)-firm est une technique de gestion de surcharge qui consiste à écarter l'exécution d'un certain nombre de jobs de tâches tout en maintenant le système dans

un état sécuritaire (sans fautes). Chaque tâche τ_i de l'ensemble est alors caractérisée par deux paramètres m_i et k_i, où m_i représente le nombre minimum de jobs qui doivent respecter leur échéance dans n'importe quelle fenêtre de k_i jobs. Une tâche sous contrainte temps réel (m, k)-firm peut se trouver dans 2 états distincts : *normal* ou *échec dynamique*. Une tâche temps réel sous contrainte (m, k)-firm est dite en état d'échec dynamique à un instant t, si à cet instant il existe plus de $(k - m)$ jobs ayant manqué leurs échéances parmi les k dernières activations de la tâche. La présence d'au moins une tâche en état d'échec met le système lui-même en état d'échec.

4.2.2 Le modèle Weakly-Hard

Le modèle avec fréquence de pertes contrainte (ou *weakly-hard* en anglais) [BBL01] est assez proche du modèle (m, k)-firm, à la différence qu'il considère des fenêtres glissantes : n jobs (éventuellement successifs) parmi m jobs successifs respectent (ou ne respectent pas) leur échéance. En ce sens, il s'agit d'une généralisation du modèle précédent. En effet, un inconvénient majeur des algorithmes basés sur le modèle (m, k)-firm est que les contraintes (m, k)-firm sont garanties seulement sur des fenêtres de temps fixées et par conséquent, des échecs dynamiques peut être rencontrés en considérant des fenêtres glissantes. De plus, ces modèles sont assez restrictifs dans le sens où ils nécessitent que toutes les tâches soit à échéances sur requêtes et possèdent la même fenêtre m. Les systèmes temps réel weakly-hard peuvent eux tolérer que certaines échéances ne soient pas respectées, pourvu que leur nombre soit borné et garanti hors-ligne.

4.2.3 Le modèle DWCS (Dynamic Window Constrained Scheduling)

Le modèle DWCS [WS99] a été défini à la base comme discipline de service pour l'ordonnancement de paquets réseau ayant pour but de maximiser le taux d'utilisation du serveur réseau en présence de multiples paquets possédant des contraintes temporelles de type (m, k)-firm. Le modèle étendu de DWCS [WGS01] est considéré comme un algorithme d'ordonnancement de tâches où chaque tâche possède une échéance relative D_i et une tolérance de pertes spécifiée par le rapport x_i de jobs pouvant être abandonnés ou retardés sur toute fenêtre de y_i jobs à exécuter pour une tâche τ_i . En cas de conflits, les jobs sont ordonnancés selon leur ordre d'arrivée.

4.2.4 Le modèle $(p + i, k)$-firm

Le modèle $(p + i, k)$-firm introduit par C. Montez et J. Fraga dans [MF02] est une extension du modèle (m, k)-firm incluant quelques unes des propriétés du calcul imprécis [CLL90]. Une tâche possède une partie obligatoire (imprécise) et une partie optionnelle (précise). Selon ce modèle, p exécutions précises et i exécutions imprécises doivent être réalisées sur k activations de la tâche. De plus, sont implémentés non seulement une attribution des priorités mais aussi un test d'acceptation de précision qui sélectionne la version précise ou imprécise de la tâche qui sera exécutée [CLL90].

4.2.5 Le modèle (m, k)-hard

La notion de contrainte (m, k)-hard a été introduite par Bernat et Burns [BB97]. Cette contrainte impose que pour chaque tâche au moins m échéances sur k activations successives doivent être respectées, sous peine d'entraîner une défaillance au niveau du système. L'approche consiste à utiliser un schéma d'exécution à double priorité (*Dual priority* [DW95]) reposant sur un algorithme à réservation de bande de faible overhead, visant à ordonnancer de façon conjointe des tâches critiques (*hard*) et des tâches non critiques (*soft*) où la capacité inutilisée du processeur est réservée à l'exécution des tâches non critiques. Cependant, cette approche nécessite

que les contraintes temporelles des tâches soient garanties par des tests d'ordonnançabilité hors-ligne.

4.2.6 Le modèle RBE

Le modèle RBE (*Rate-Based Execution*) [JG99] considère des contraintes exprimées en termes d'unités de temps plutôt qu'en termes d'échéances. Une tâche RBE est ainsi caractérisée par un 4-tuple (x_i, y_i, d_i, c_i) où :

- x_i est le nombre maximum de jobs susceptibles d'être réveillés sur tout intervalle de longueur y_i ,
- y_i est un intervalle de temps,
- d_i est l'échéance relative de la tâche,
- c_i est sa durée d'exécution dans le pire cas.

Le job J_{ij} d'une tâche réveillé à t_{ij} doit s'exécuter avant une échéance $D_i(j)$ donnée par la relation de récurrence suivante :

$$D_i(j) = \begin{cases} t_{ij} + d_i & si\ 1 \leq j \leq xi \\ \max\left(t_{ij} + d_i, D_i(j - x_i) + y_i\right) & si\ j > x_i \end{cases} \tag{1.15}$$

Sous l'hypothèse de l'utilisation de cette fonction d'affectation d'échéance dans le modèle RBE, Jeffay and Godard montrent dans [JG99] que EDF est une politique optimale pour l'ordonnancement préemptif, l'ordonnancement non-préemptif, et l'ordonnancement en présence de ressources partagées.

4.2.7 Le modèle MC (Markov Chain)

Hu *et al.* dans [HLLL03] propose un modèle tolérant aux pertes basé sur une chaîne de Markov (*Markov Chain*) pour décrire le comportement stochastique désiré en termes de pertes associées à une tâche τ_i. On dénote une MC-contrainte de τ_i par MC_i. MC_i est un processus stochastique discret possédant 2 états ou plus. La probabilité de transition d'un état à un autre représente soit la probabilité pour le prochain job d'être abandonné, soit sa probabilité de réussite. Chaque état est représenté par une chaîne spécifique de bits d'état f_{ij} traduisant le comportement des jobs précédents : un '1' pour un job réussi, un '0' pour un job abandonné.

4.2.8 Le modèle Skip-Over

Dans ce modèle, chaque tâche périodique $\tau_i(C_i, T_i, s_i)$ est caractérisée par une durée d'exécution au pire-cas C_i, une période T_i, une échéance relative égale à sa période, et un paramètre de pertes s_i, $2 \leq s_i \leq \infty$, qui fournit la tolérance de la tâche à manquer ses échéances. Selon la terminologie introduite par Koren et Shasha dans [KS95], chaque job de tâche peut être *rouge* (*red*) ou *bleu* (*blue*). Un job rouge doit être exécuté avant son échéance ; un job bleu peut être abandonné à tout moment. Un ensemble de tâches est dit profondément rouge (*deeply-red*, en anglais) si l'ensemble des jobs réveillés à $t = 0$ sont rouges. Ce modèle doit suivre un certain nombre de règles élémentaires parmi lesquelles :

- les $s_i - 1$ premiers jobs de chaque tâche sont rouges.
- une perte survient lorsqu'un job bleu ne termine pas son exécution avant sa date d'échéance.
- la distance entre 2 pertes consécutives doit être d'au moins s_i périodes.
- si un job bleu est abandonné, les $s_i - 1$ prochains jobs de la tâche sont rouges.
- si un job bleu s'exécute en respectant son échéance, le prochain job de la tâche reste bleu.

Ce modèle est un cas particulier du concept $(m, k) - firm$ où $m = k - 1$.
Lorsque $s_i = \infty$, aucune perte n'est autorisée, ce qui équivaut au modèle classique d'une tâche périodique temps réel à contraintes strictes.

Deux algorithmes d'ordonnancement décrits dans [KS95] dérivent de ce modèle. Le premier algorithme est l'algorithme *RTO* (*Red Tasks Only*) dans lequel les jobs bleus sont systématiquement

rejetés et les jobs rouges sont ordonnancés selon EDF. Cet algorithme assure une QoS minimale en ne respectant que les échéances des jobs rouges. La distance entre les pertes est ainsi constante et égale à s_i périodes. Un autre algorithme nommé *BWP* est lui plus flexible du fait qu'il autorise l'exécution des jobs bleus lorsque cela ne compromet pas l'exécution des jobs rouges. Cet algorithme exécute donc les jobs rouges selon EDF et s'il n'y en a aucun prêt, il ordonnance alors un job bleu. S'il y a plus d'un job bleu alors il en élit un selon une heuristique donnée, par exemple celle dont l'échéance est la plus proche.

À titre d'illustration, considérons l'exemple de la Figure 1.13 sur une plateforme monoprocesseur :

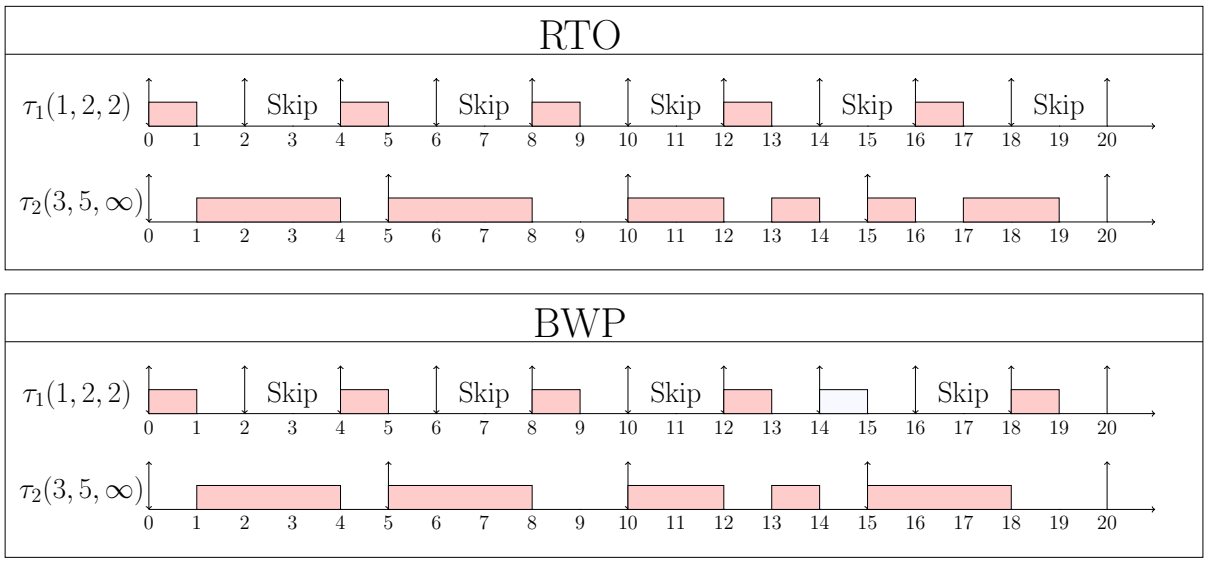


FIGURE 1.13 – Exemple d'ordonnancement au sens Skip-Over

Il y a surcharge ($U_p = \frac{1}{2} + \frac{3}{5} = 1.1$) mais le fait d'autoriser certaines pertes va permettre à l'ensemble de tâches d'être ordonnançable.

Koren et Shasha ont prouvé que le problème de faisabilité d'un ensemble de tâches périodiques autorisant des pertes au sens Skip-Over est NP-difficile [KS95]. En d'autres termes, pour trouver un ensemble ordonnançable, nous devons énumérer toutes les séquences d'ordonnancement possibles. Pour cela, Koren et Shasha ont donné dans un premier temps la condition nécessaire de faisabilité suivante :

Théorème 15 *Un ensemble $\tau = \{\tau_i(C_i, T_i, s_i), i = 1...n\}$ de n tâches périodiques autorisant des pertes au sens Skip-Over est ordonnançable seulement si :*

$$\sum_{i=1}^{n} \frac{Ci(s_i - 1)}{T_i} \leq 1 \tag{1.16}$$

Caccamo et Buttazo [CB97] ont introduit la notion de *facteur équivalent* représentant la charge de traitement imposée au processeur pour un fonctionnement avec une qualité de service minimale (seuls les jobs rouges sont exécutés). Cette notion est définie comme suit :

Définition 1.24 *Étant donné un ensemble $\tau = \{\tau_i(C_i, T_i, s_i), i = 1..n\}$ de n tâches périodiques autorisant des pertes au sens Skip-Over, le facteur d'utilisation équivalent du processeur est défini par :*

$$U_p^* = \max_{L \geq 0} \frac{\sum_{i=1}^{n} D(i, [0, L])}{L} \tag{1.17}$$

où

$$D(i,[0,L]) = \left(\lfloor \frac{L}{T_i} \rfloor - \lfloor \frac{L}{s_i T_i} \rfloor \right) \tag{1.18}$$

Ils ont prouvé dans [CB98] qu'une condition suffisante garantissant la faisabilité d'un ensemble de tâches tolérantes aux pertes au sens Skip-Over est la suivante :

Théorème 16 *Un ensemble de tâches périodiques tolérants aux pertes au sens Skip-Over est faisable si* $U_p^* \leq 1$.

Cette condition se révèle nécessaire et suffisante dans le cas d'ensembles de tâches profondément rouges.

Ce modèle possède l'avantage de prendre en compte la consécutivité des violations d'échéances de manière claire et de permettre la modélisation de tâches à contraintes strictes ($s_i = \infty$)***. Nous utiliserons dans nos travaux ce modèle pour gérer les cas de surcharge.***

5 Ordonnancement temps réel et gestion de l'énergie

5.1 Vers des systèmes autonomes : réseaux de capteurs sans fil

L'alimentation en énergie électrique devient une problématique cruciale en particulier dans la conception des systèmes nomades qui par nature doivent être autonomes du point de vue énergétique tels que les réseaux de capteurs sans fils (*Wireless Sensor Network*, en anglais). Leur concept est apparu à partir de la fin des années 70, à l'Université de Carnegie-Mellon où ont émergé des recherches sur ce domaine financées par le Département de la Défense des États-Unis. Un réseau de capteurs sans fil introduit une nouvelle technologie d'acquisition et de traitement des données, qui intègre trois principales fonctionnalités dans un même système : la technologie de capteur, la technologie MEMS (*Micro-Electro-Mechanical System*) et le réseau sans fil. Un capteur sans fil constitue un système embarqué avec des caractéristiques temps réel. Les capteurs sans fil sont souvent destinés à relever des informations en environnements hostiles auxquels l'homme n'a pas toujours accès. Une fois déployés, les capteurs deviennent entièrement autonomes. Cette autonomie est cependant dépendante de la (ou les) batterie(s) qui les alimente(nt). La durée de vie et la durée de charge de celles-ci détermineront donc la durée de vie du système. Les concepteurs se préoccupent donc principalement de l'aspect énergétique de manière à pouvoir fournir la plus longue autonomie aux capteurs tout en respectant les contraintes liées à l'alimentation du système. Cette problématique est d'autant plus un challenge que les applications deviennent plus sophistiquées : plus de traitements, plus de communications et donc davantage d'énergie consommée. Ce constat a poussé certains concepteurs à s'orienter vers des systèmes puisant leur énergie de l'environnement en fonction des besoins plutôt que d'embarquer l'énergie nécessaire pour toute la durée de vie d'une application. On fera référence à cette catégorie de systèmes par son abréviation en anglais **EH-RTES** (*Energy Harvesting-Real-time Embedded Systems*) [Yil11] dans la suite du manuscrit. En fonction des besoins du système en énergie, on préferera certaines sources d'énergie à d'autres. Le tableau suivant représente l'intérêt du choix de la source d'énergie en fonction des besoins en puissance de l'application.

Source d'énergie	Densité de puissance et performance	Source d'information
Bruit acoustique	$0.003\mu W/cm^3$ @ 75Db $0.96\mu W/cm^3$ @ 100Db	[RADS+00]
Variation de température	$10\mu W/cm^3$	[RSF+04]
Lumière ambiante	$100mW/cm^2$ (soleil) $100\_W/cm^2$ (bureau éclairé)	disponible
Onde radioélectrique ambiante	$1\mu W/cm^2$	[Yea04]
Thermoélectricité	$60\_W/cm^2$	[Ste99]
Vibration (micro générateur)	$4\_W/cm^3$ (mouvement humain - Hz) $800\_W/cm^3$ (machines - kHz)	[MGYH04]
Vibrations (piezoélectriques)	$200\mu W/cm^3$	[RWP02]
Flux d'air	$1\mu W/cm^2$	[Hol04]
Boutons poussoirs	$50\_J/N$	[PF01]
Semelles piézo-électriques	$330\mu W/cm^2$	[SP01]
Générateurs à manivelle	$30W/kg$	[SP04]
Piézoélectricité (frappe du talon)	$7W/cm^2$	[Yag02, SP01]

TABLE 1.1 – Comparaison des densités de puissance de différentes méthodes de récupération d'énergie [Yil11]

5.2 Stratégies d'ordonnancement temps réel sur des plateformes autonomes du point de vue énergétique

Il devient important de s'intéresser à la conception de politiques de gestion de l'énergie pour concevoir des systèmes entièrement autonomes. Du point de vue recherche, une thématique intéressante repose sur l'étude de politiques d'ordonnancement de tâches temps réel sous contraintes énergétiques. Il existe une littérature pour ce type de problèmes, mais elle se cantonne principalement au cas monoprocesseur. Les techniques existantes dans le domaine de la gestion d'énergie des systèmes temps réel embarqués récupérant l'énergie de l'environnement s'appuient sur (i) soit des techniques basées sur la variation de la vitesse de fonctionnement du processeur appelées DVFS (*Dynamic Voltage Frequency Scaling*) [LQW08, LQW09], (ii) soit des techniques de gestion dynamique de la mise sous/hors tension appelées DPM (*Dynamic Power Management*) [MBTB06].

Dans [AM01], Allavena et al. décrivent le problème qui consiste à déterminer une séquence valide d'ordonnancement d'un ensemble de tâches périodiques indépendantes à périodes identiques, en supposant que le réservoir d'énergie (par exemple une batterie) est initialement rempli. Toute séquence valide se termine avec un niveau d'énergie du réservoir au moins égal au niveau initial. Leur ordonnanceur (hors-ligne) utilise une sélection de la tension et de la fréquence du processeur du système pour réduire la consommation énergétique en ralentissant l'exécution des tâches sous contraintes temporelles. Leur approche repose cependant sur l'hypothèse que les puissances de consommation et de production instantanées sont constantes. Dans [MBTB06], les tâches sont exécutées au plus tard à pleine vitesse en utilisant un algorithme d'ordonnancement appelé LSA(*Lazy Scheduling Algorithm*). Cet algorithme est optimal et, comparé à l'ordonnancement EDF, il conserve le même taux d'échéances manquées mais avec une batterie de plus petite capacité (plus petite de 25% comparé à EDF). Un autre algorithme nommé EA-DVFS (Energy-Aware Dynamic Voltage et Frequency Selection) a été proposé dans [LQW08]. Il économise de l'énergie en ralentissant les tâches lorsque l'énergie stockée n'est pas suffisante. Le but de EA-DVFS est d'utiliser efficacement la laxité des tâches de manière à réduire le taux d'échéances manquées. Les processeurs doivent choisir entre une puissance maximale ou réduite de calcul, en fonction de l'énergie disponible. Si le système possède suffisamment d'énergie, la tâche est exécutée à pleine vitesse ; sinon, elle est étirée et exécuté à une vitesse inférieure. En cas de faible charge de travail, l'algorithme EA-DVFS réduit le taux d'échéances manquées de 50% par rapport à LSA et réduit la taille minimale de stockage de 25% lorsque le taux d'échéances manquées est nul. Cependant, il est difficile de régler le voltage lorsqu'on est en forte surcharge et, par conséquent, le taux d'échéances manquées devient très élevé. Une extension de EA-DVFS a ensuite été présentée dans [LQW09] où les auteurs utilisent un ordonnancement adaptatif avec

l'algorithme DVFS nommé AS-DVFS (*Adaptive Scheduling and DVFS algorithm*), dans lequel le réglage de la fréquence de fonctionnement du processeur tient compte des contraintes temporelles et des contraintes d'énergie. L'efficacité énergétique est atteinte en exploitant les laxités des tâches pour récupérer de l'énergie en initialisant la séquence d'ordonnancement selon LSA puis en distribuant uniformément la charge de travail au fil du temps. Les techniques énoncées ci-dessus fonctionnent cependant exclusivement sur des plateformes monoprocesseur.

Parallèlement, l'ordonnancement sous contraintes énergétiques pour des systèmes temps réel embarqués multiprocesseur n'a pas reçu beaucoup d'attention. Aydin *et al.* [AY03] ont montré que l'ordonnancement multicœur est un problème NP-difficile, et ont développé une technique ayant pour objectif la minimisation de la consommation énergétique en utilisant l'algorithme EDF avec variation de la tension du processeur. Dans [ZYTT09], les auteurs ont proposé un algorithme de partitionnement AMBFF (*Adaptive Minimal Bound First-Fit*) tenant compte des contraintes de manière réaliste afin d'économiser plus d'énergie. Les auteurs de [CT10] ont proposé un nouvel algorithme d'approximation destiné à la réduction de la consommation énergétique pour une plateforme multiprocesseur homogène avec une facteur d'approximation de 1.21. Dans [LQ11], les auteurs ont proposé une nouvelle technique de partitionnement basé sur l'algorithme DVFS pour des EH-RTES multicœur. Par ailleurs, le travail dans [WGCH10] introduit un régime d'allocation des tâches aux processeurs visant à réduire la consommation énergétique d'un système EH-RTES multiprocesseur sur puce ou SoC (*System-on-Chip*, en anglais). Les auteurs ont proposé un schéma d'allocation hors-ligne des tâches aux processeurs de manière à assurer une consommation énergétique adaptée aux contraintes du système. La consommation énergétique des tâches est ensuite réduite en équilibrant la charge de travail de l'application et en exécutant les tâches au plus tard sans compromettre l'ordonnançabilité du système et le respect des contraintes de précédence.

À l'heure actuelle, aucune étude n'a considéré l'adaptation simple des différentes heuristiques de partitionnement existantes (FF, BF, WF, NF) à des EH-RTES multicœur.

Chapitre 2

Contribution au partitionnement

Ce chapitre est consacré au problème relatif au partitionnement d'un ensemble de tâches temps réel périodiques à contraintes strictes sur une plateforme multiprocesseur homogène. Nous exposons en particulier une nouvelle approche de partitionnement basée sur le fractionnement de tâches (task splitting). Les performances de cette méthode sont comparées à celles d'un algorithme basé sur la technique de task splitting : ***EDF Split*** *(DD) dont l'approche est décrite par la suite.*

1 Modèle du système considéré

Dans ce chapitre, nous considérons une plateforme π composée de m processeurs identiques : $\pi = \{\pi_1, ..., \pi_m\}$, et un ensemble de n tâches périodiques temps réel dur $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Chaque tâche $\tau_i\,(O_i, C_i, T_i, D_i)$ est définie par :

- O_i, l'offset de τ_i ;
- C_i, la durée d'exécution pire-cas de τ_i ;
- T_i, la période de la tâche τ_i ;
- D_i, l'échéance relative de la tâche τ_i ;

Le facteur d'utilisation ou charge processeur de la tâche τ_i dénoté u_i correspond au taux d'activité du processeur dédié à l'exécution des jobs successifs de la tâche : $u_i = \frac{C_i}{T_i}$. Le facteur d'utilisation d'un système composé de m processeurs et n tâches périodiques est défini comme suit :

$$u_{sys} = \frac{1}{m} \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.1}$$

Chaque tâche τ_i génère un nombre infini de jobs $\tau_{i,j}$, $j \geq 0$. Chaque job possède une date de réveil $r_{i,j} = O_i + j.T_i$ et une échéance absolue $d_{i,j} = r_{i,j} + D_i$. L'ensemble de tâches est dit synchrone lorsque l'ensemble des offsets des tâches sont nuls ; dans le cas contraire, il est dit asynchrone. Les tâches considérées sont préemptibles et indépendantes. Nous nous intéressons au modèle de tâches à échéances contraintes ($D_i \leq T_i$).

Dans ce système, nous nous intéressons à l'approche par partitionnement. Le partitionnement est une opération hors-ligne consistant à déterminer la répartition de l'ensemble des tâches de la configuration sur les différents processeurs, de manière à ce que la configuration globale soit ordonnançable. Sur chacun des processeurs, il convient de vérifier l'ordonnançabilité du sous-ensemble de tâches alloué. Plus formellement, il s'agit de partitionner l'ensemble des n tâches en m sous-ensembles disjoints $\Gamma_1, \Gamma_2, ..., \Gamma_m$ (avec $\bigcup \Gamma_j = \tau$) et d'ordonnancer ensuite chaque sous-ensemble Γ_j sur le processeur π_j avec une politique d'ordonnancement monoprocesseur. Aucune migration de tâches n'est autorisée.

Dans la suite, nous présentons et illustrons le principe d'un algorithme basé sur la technique de *task splitting*, technique de fractionnement de tâches que nous avons également retenue pour la proposition de notre nouvelle approche de partitionnement.

2 Technique de task splitting : exemple de l'approche C=D

2.1 Principe du task splitting

Une limitation inhérente aux schémas d'ordonnancement partitionné repose sur le fait que chaque tâche doit s'exécuter entièrement sur un processeur donné. Cette contrainte rend la détermination d'un partitionnement optimal NP-difficile [GJ90] et empêche tout algorithme d'ordonnancement partitionné pur d'atteindre un seuil d'utilisation du système supérieur à $(m+1)/2$ [ABJ01] pour des ensembles composés de tâches périodiques à échéances sur requêtes.

Des recherches récentes se sont attaquées à ce problème de limitation en développant des techniques de task splitting. Ces algorithmes dévient du modèle de partitionnement strict dans le sens où une tâche peut être amenée à s'exécuter sur plusieurs processeurs. La stratégie de partitionnement alloue toujours les tâches aux différents processeurs selon une heuristique donnée. Cependant, lorsque l'algorithme de partitionnement atteint un point pour lequel l'ajout d'une nouvelle tâche à un processeur donné conduit à la non-ordonnançabilité du sous-ensemble de tâches alloué sur ce processeur, il fractionne la tâche en deux parties, en allouant une fraction au processeur donné de telle sorte que les contraintes temporelles des tâches soient respectées, et la fraction restante de la tâche à un autre processeur.

Cette approche permet ainsi de passer outre la limitation du $(m+1)/2$ sur le facteur d'utilisation du système (voir 2.2.5 Exemple illustratif).

Nous décrivons à présent le fonctionnement de l'algorithme EDF Split (DD) basé sur l'approche de task splitting $C = D$.

2.2 L'approche C=D

Dans leurs travaux [BDWZ12], les auteurs considèrent un modèle de tâches périodiques synchrones. Chaque tâche est caractérisée par le 3-tuple (C_i, T_i, D_i).

L'approche $C = D$ est basée sur le concept de semi-partitionnement et consiste en 2 phases :

1. Une phase de partitionnement *hors-ligne* qui consiste à déterminer la répartition de l'ensemble des tâches sur les différents processeurs de telle sorte que chacun des sous-ensembles de tâches alloués sur les processeurs soient ordonnançables,
2. Une phase d'ordonnancement *en-ligne* qui consiste à ordonnancer localement les tâches sur les différents processeurs en respectant les contraintes de précédence introduites par le *task splitting* réalisé dans la phase (1).

2.2.1 La phase de partitionnement

Les tâches sont d'abord triées selon une certaine heuristique de Bin-packing basée, par exemple, sur l'utilisation ou la densité associée à la tâche.

- Chaque processeur (π_j) *reçoit* des tâches jusqu'à ce que l'ajout d'une nouvelle tâche soit tel que l'ordonnançabilité de l'ensemble ne puisse pas être vérifiée.
- La tâche suivante $\tau_s(C_s, T_s, D_s)$ est alors fractionnée de telle sorte que la première partie soit allouée au processeur π_j et que l'ensemble ainsi constitué soit ordonnançable.
- La première partie de la tâche fractionnée (τ_s^1) présente la contrainte suivante : son échéance relative est réduite pour être égale au temps d'exécution maximum tel que la tâche puisse être logée sur π_j. Son profil est donc $(C_s^1, T_s, D_s^1 = C_s^1)$.
- La deuxième partie de la tâche fractionnée (τ_s^2) a le profil suivant $(C_s^2 = C_s - C_s^1, T_s, D_s^2 = D_s - D_s^1)$. Elle est attribuée au processeur π_{j+1}.
- Le partitionnement se poursuit par l'attribution de nouvelles tâches au processeur π_{j+1} jusqu'à ce processeur ne puisse plus à son tour accueillir aucune tâche complète. Une autre tâche est alors fractionnée entre le processeur π_{j+1} et π_{j+2}.

Au plus $m-1$ tâches sont fractionnées selon l'approche $C = D$.

2.2.2 La phase d'ordonnancement

L'approche *C=D* introduit, de par le découpage même des tâches, des contraintes de précédence sur les portions de tâches découpées qu'il convient de respecter.

Il s'en suit que pour une tâche fractionnée τ_s :

- La première partie de la tâche fractionnée τ_s^1 s'exécute de manière non-préemptive de t à $t + D_s^1$ sur le processeur π_j selon sa durée d'exécution $C_s^1 = D_s^1$.
- À l'instant $t + D_s^1$, l'exécution de la deuxième partie de la tâche bascule sur le processeur π_{j+1}. La tâche τ_s^2 s'exécute alors de manière préemptive sur π_{j+1} et possède une échéance absolue égale à $t + D_s^1 + D_s^2$ soit $t + D_s$, l'échéance de la tâche d'origine non fractionnée.

Il en résulte que les deux parties de la tâche ne s'exécutent jamais simultanément. La dépendance d'exécution entre τ_s^1 et τ_s^2 implique au niveau du système d'exploitation temps réel, de disposer d'un timer assez précis pour mesurer la durée d'exécution de τ_s^1 sur π_j et ainsi activer au bon moment l'exécution de τ_s^2.

2.2.3 Détermination des paramètres des tâches fractionnées

Le calcul de la valeur de C_s^1 s'appuie sur une analyse de sensibilité [BB05] qui est une généralisation de l'analyse de faisabilité. En effet, l'analyse de sensibilité peut être appliquée à un modèle de système dans lequel les durées d'exécution des tâches sont définies avec un certain degré d'incertitude.

Dans le cas présent, si l'ensemble de tâches alloué au processeur π_j n'est pas ordonnançable, l'analyse de sensibilité fournit une indication quantitative au niveau des actions requises pour ramener la configuration dans un état où l'ordonnançabilité sera vérifiée (i.e. abaissement de la charge associée à la tâche en raccourcissant sa durée d'exécution sur le processeur π_j).

Pour utiliser la méthode $C = D$, les auteurs appliquent l'analyse de sensibilité pour un système de tâches ordonnancées localement par EDF. L'algorithme QPA (Quick Processor-demand Analysis) utilisé est décrit dans [ZB08a, ZB08b].

En supposant qu'il y ait violation d'échéance(s) sur le processeur π_j à partir du moment où la tâche τ_s est ajoutée, les étapes suivantes sont effectuées :

1. Choisir (initialement) C_s^1 ($< C_s$) de telle sorte que l'utilisation du processeur π_j soit égale à 1.
2. Définir $D_s^1 = C_s^1$.
3. Calculer L, de manière à trouver l'intervalle de charge maximale du processeur $h(t)$.
4. Appliquer l'algorithme QPA sur $[0, L]$.
5. S'il y a une violation d'échéance (la demande du processeur maximale est supérieure à 1), réduire C_s^1 (et par la suite D_s^1).
6. Si la nouvelle valeur de C_s^1 est 0 alors aucune partie de la tâche ne peut être logée sur le processeur π_j.
7. Sinon, s'il n'y a aucun échec avec la valeur actuelle de C_s^1 alors celle-ci est la valeur optimale de temps logé sur π_j.

Supposons qu'il y ait violation d'échéance à un instant t, $h(t) > t$. La valeur de C_s^1 doit alors être réduite. La valeur recalculée de C_s^1 découle directement de la demande du processeur au moment de la faute. Dans l'intervalle $[0, t[$, la quantité de temps nécessaire à l'exécution de toutes les autres tâches, $Oth(t)$, est donnée par :

$$Oth(t) = \sum_{\substack{\tau_j \in \pi_j \\ \tau_j \neq \tau_s}} \lfloor \frac{t + T_j - D_j}{T_j} \rfloor C_j \tag{2.2}$$

où $Oth(t)$ représente la somme des temps d'exécution des jobs de tâches assignées au processeur π_j (excepté τ_s). Le reste du temps sera réservé à l'exécution de la première partie de τ_s. En

d'autres termes, le temps d'exécution de la première partie, C_s^1, de τ_s a une valeur maximale définie comme suit :

$$C_s^1 = (t - Oth(t)) / \lfloor \frac{t + T_s - D_s^1}{T_s} \rfloor \tag{2.3}$$

D_s^1 étant égale à C_s^1 :

$$C_s^1 = (t - Oth(t)) / \lfloor \frac{t + T_s - C_s^1}{T_s} \rfloor \tag{2.4}$$

La valeur de C_s^1, l'inconnu de cette formule est extraite par récurrence de la manière suivante :

$$C_s^1(r+1) = (t - Oth(t)) / \lfloor \frac{t + T_s - C_s^1(r)}{T_s} \rfloor \tag{2.5}$$

sachant que la valeur initiale $C_s^1(1)$ est calculée de manière à ce que l'utilisation du processeur π_j soit égale à 1.

2.2.4 Algorithme EDF Split (DD)

EDF Split (DD) [BDWZ12] est un algorithme reposant sur le principe de l'approche $C = D$. Il attribue les tâches aux processeurs selon l'heuristique FFDD (*First-Fit Decreasing Density*), puis ordonnance les tâches assignées à chaque processeur (fractionnées ou non) selon EDF.

À partir du moment où aucune autre tâche ne peut être attribuée au premier processeur π_j sans causer une violation d'échéance, l'une des tâches restantes (non-assignées) possèdant la plus grande densité est alors divisée entre le processeur π_j et π_{j+1}. La répartition des tâches se poursuit par le processeur π_{j+1} dans l'ordre décroissant de leur densité et ainsi de suite. L'efficacité de cet algorithme vis-à-vis des approches purement partitionnées a été évaluée dans [BDWZ12].

Les résultats présentés montrent que l'approche $C = D$ offre un meilleur taux d'utilisation du système. Cependant, le coût des migrations (même si celui-ci est nettement réduit par rapport à une approche globale) n'est pas quantifié. De même, les auteurs n'évoquent pas l'overhead associé à la synchronisation des portions des tâches fractionnées.

2.2.5 Exemple illustratif

Considérons un ensemble $\tau = \{\tau_1(70, 100, 100), \tau_2(15, 25, 25), \tau_3(25, 50, 50)\}$. Le facteur d'utilisation de cet ensemble de tâches est égal à 1.82. Une approche de partitionnement classique nécessiterait cependant trois processeurs (un pour chaque tâche) alors que l'approche $C = D$ n'en nécessite que deux. Le tableau 2.1 contient le détail de l'allocation des tâches aux différents processeurs.

τ_i	C_i	T_i	D_i	π_j
τ_1	70	100	100	π_1
τ_2^1	7	25	7	π_1
τ_2^2	8	25	18	π_2
τ_3	26	50	50	π_2

TABLE 2.1 – Exemple du résultat de partitionnement de l'ensemble de tâches τ

La tâche τ_2 exécute sa première partie τ_2^1 sur le premier processeur avec la tâche τ_1. La seconde partie de la tâche τ_2 est activée à l'instant $t = 7$ et possède un temps d'exécution égal à 8 et une échéance relative de 18. Celle-ci s'exécute sur le second processeur avec la tâche τ_3.

Malgré un facteur d'utilisation de τ égal à 1.82 ($> (m+1)/2 = 1.5$) et des tâches possèdant toutes un facteur d'utilisation u_i supérieur à 0.5, EDF Split arrive à ordonnancer correctement

l'ensemble τ. Cet exemple montre donc comment EDF Split peut passer outre la limitation du $(m+1)/2$ sur le facteur d'utilisation du système.

Dans la suite, sur la base du principe de *task splitting*, nous introduisons une nouvelle approche de partitionnement qui offre l'avantage de n'induire aucune migration de tâches. Nous montrons ainsi qu'il est possible d'atteindre des performances similaires (voire meilleures dans certains cas) que sous l'algorithme EDF Split (DD) en s'affranchissant des coûts de synchronisation des portions de tâches fractionnées.

3 Proposition d'une approche de partitionnement de type task splitting

Le modèle de tâches considéré est celui présenté précédemment dans la section 1, à savoir des tâches périodiques asynchrones à échéances contraintes.

3.1 Principe

L'objectif de l'approche KTS (*K-level Task Splitting*) est de fournir une solution purement partitionnée reposant sur le principe de *task splitting* et pour laquelle le facteur d'utilisation du système sera aussi élevé que celui observé par une approche de semi-partitionnement. Comme pour toute approche partitionnée, les tâches sont d'abord assignées aux différents processeurs selon une certaine heuristique de base (par exemple, FFDU : First Fit Decreasing Utilization, BFDD : Best Fit Decreasing Density) basée, par exemple, sur l'utilisation ou la densité. Dès lors que l'allocation d'une tâche sur un processeur n'est pas possible, la tâche rejetée va subir un fractionnement selon l'algorithme KTS, comme le montre la Figure 2.1.

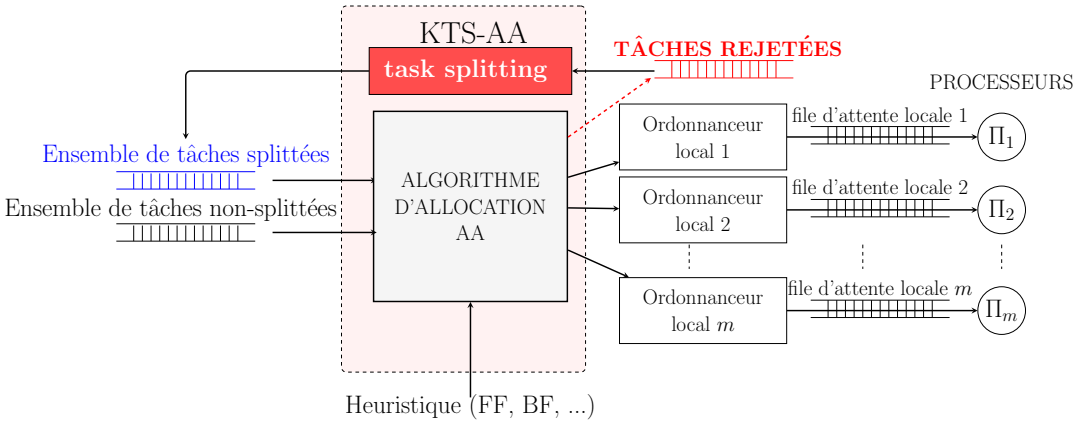


FIGURE 2.1 – Schéma de fonctionnement de KTS

Lorsqu'une tâche τ_s ne peut plus être ajoutée à aucun processeur sans compromettre l'ordonnançabilité de l'ensemble de tâches alloué sur celui-ci, cette tâche, τ_s est alors fractionnée en deux nouvelles tâches indépendantes asynchrones. Ces dernières présentent un facteur d'utilisation moindre (soit la moitié de la valeur du facteur d'utilisation de la tâche d'origine τ_s) et intégrent des offsets de manière à ce qu'elles ne s'exécutent jamais simultanément.

Définition 2.1 *Une tâche $\tau_s(O_s, C_s, T_s, D_s)$ rejetée donne lieu à un nouvelle ensemble τ_s^{split} composé de 2 tâches indépendantes ayant les caractéristiques suivantes : $\tau_s^{split} = \{\tau_s^x(O_s + x \times T_s, C_s, 2 \times T_s, D_s), x = 0..1\}$*

Considérons l'exemple de la Figure 2.2 illustrant le fractionnement spécifié dans la définition précédente. Une tâche $\tau_s(0, 2, 4, 3)$ ne pouvant être assignée à aucun processeur est splittée

en deux tâches indépendantes $\tau_s^0(0,2,8,3)$ et $\tau_s^1(4,2,8,3)$ possèdant chacune le même temps d'exécution que τ_s mais dont la période a été doublée. De plus, la tâche τ_s^1 possède un offset de manière à ce qu'il n'y ait pas de chevauchement entre les différents jobs de τ_s^0 et τ_s^1.

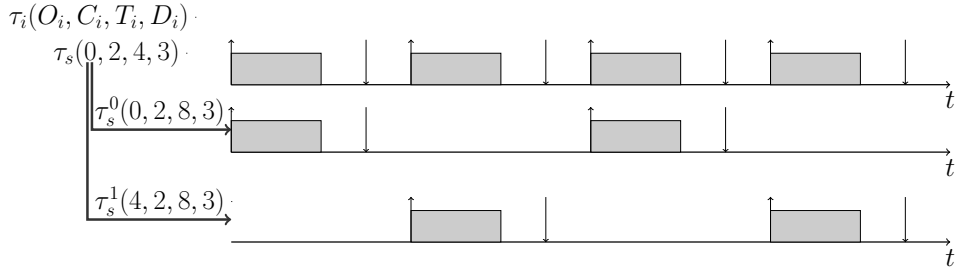


FIGURE 2.2 – Splitting d'une tâche τ_s

L'algorithme essaie ensuite d'attribuer chacune des deux tâches créées à un processeur. Le test exact d'ordonnançabilité utilisé est celui décrit dans le Théoreme 10 (page 19). Celui-ci va permettre de déterminer si un ensemble de tâches périodiques asynchrones peut être ordonnancé correctement selon EDF sur un processeur donné π_j. Si une tâche splittée ne peut être attribuée à aucun processeur, elle subit de nouveau une étape de fractionnement (*task splitting*). À chaque étape de task splitting correspond un niveau. On définit une profondeur limite de *task splitting* K, c'est-à-dire que chaque tâche de l'ensemble ne peut être splittée plus de K fois. La Figure 2.3 illustre l'algorithme pour une limite $K = 2$, ce qui veut dire que l'on autorise jusqu'à deux niveaux de *task splitting* pour chaque tâche. Au niveau 0, la tâche τ_3 ne peut être assignée à aucun processeur. Elle est alors fractionnée en deux autres tâches indépendantes τ_3^0 et τ_3^1 à un premier niveau de *task splitting*. De ces deux nouvelles tâches, l'une (τ_3^0) est assignée correctement à un processeur alors que l'autre (τ_3^1) est rejetée par l'ensemble des processeurs. Celle-ci est alors fractionnée en deux autres tâches au niveau 2 de *task splitting*. Cet algoritme continue ainsi jusqu'à ce qu'il atteigne la profondeur limite de task splitting ou que l'ensemble des tâches ait été correctement assigné à l'ensemble des processeurs. Dans l'exemple de la Figure 2.3, l'algorithme atteint le niveau de profondeur limite $K = 2$ dans lequel l'ensemble des tâches est affecté avec succès. Par conséquent, l'ensemble de tâches est ordonnançable sur la plateforme donnée.

Il est toutefois important de noter que l'algorithme KTS avec $K = 0$ est équivalent à l'heuristique de partitionnement classique.

Malgré le fait qu'avec KTS une tâche d'origine puisse s'exécuter sur différents processeurs suite à son fractionnement, cette approche s'apparente à une approche de partitionnement plutôt qu'une approche de semi-partitionnement, dans le sens où :

1. Chaque tâche splittée (fractionnée) est considérée comme une nouvelle tâche périodique complètement indépendante affectée de manière unique à un processeur, au contraire de certaines méthodes globales ou semi-partionnées où la migration peut se faire au niveau du job de la tâche en fonction de la disponibilité des processeurs.

2. Le code de la tâche d'origine est dupliqué sur les différents processeurs pour chacune des deux tâches splittées. Comme les jobs d'une tâche donnée sont supposées indépendants les uns des autres dans la plupart des techniques d'analyse de WCET (par exemple, chaque job est généralement supposé commencer avec un cache vide), la migration des données n'est donc pas nécessaire.

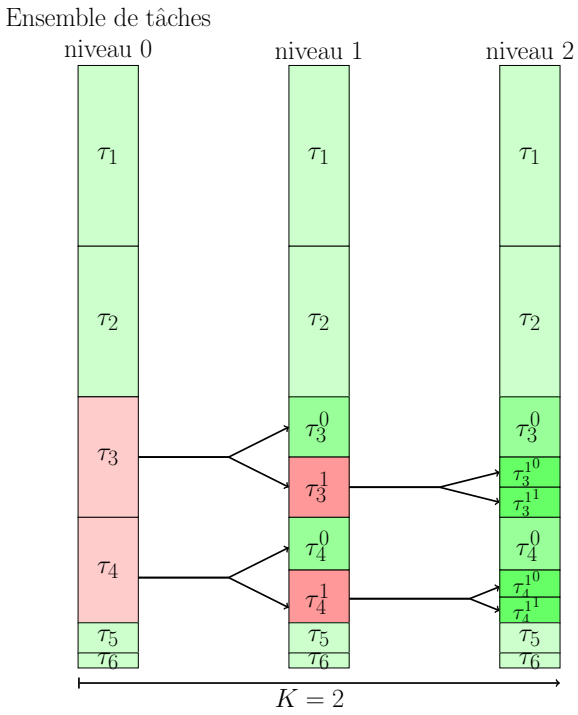


FIGURE 2.3 – *Task splittings* consécutifs de tâches selon le paramètre K de l'algorithme KTS

La seule exigence d'un tel algorithme est que toutes les horloges des processeurs soient synchronisées.

3.2 Description algorithmique

La Figure 2.4 montrent le pseudo-code de l'algorithme KTS utilisé avec l'heuristique FF, et dénommé ci-après KTS-FF. k représente le niveau de *task splitting* courant atteint et $maxK$ est la profondeur limite de *task splitting* (notée précédemment par K). Sur une plateforme multiprocesseur, $\tau(\pi)$ désigne l'ensemble de tâches parmi τ_1,..., τ_{i-1} ayant été assignées avec succès à l'un des processeurs de l'ensemble π. Initialement, $\tau(\pi) = \varnothing$. Au début, l'algorithme fonctionne selon une heuristique de partitionnement donnée. Si aucun processeur n'est capable de recevoir une certaine tâche, nous ne pouvons rien conclure quant à l'ordonnançabilité de l'ensemble de tâches τ sur la plateforme multiprocesseur. La tâche τ_i est alors fractionnée en un sous-ensemble τ_i^{split} composé de deux tâches qui sont assignées à un sous-ensemble de processeurs de la plateforme à m processeurs. Le Lemme 2.1 affirme qu'en attribuant une tâche fractionnée de τ_i^{split} à un processeur π_j, l'ordonnançabilité des tâches confiées précédemment à l'ensemble des processeurs reste intacte.

Lemme 2.1 *Si l'ensemble de tâches précédemment assignées aux processeurs est faisable (sur chaque processeur) et que par la suite l'algorithme KTS assigne une tâche τ_i^0 (respectivement τ_i^1) au processeur π_j (d'après le Théoreme 10), alors l'ensemble de tâches assignées à chaque processeur (y compris processeur π_j) reste faisable.*

Preuve: On observe que l'ordonnançabilité des processeurs autres que le processeur π_j n'est pas affectée par l'allocation de la tâche τ_i^0 (respectivement τ_i^1) au processeur π_j. Aussi, si l'ensemble de tâches précédemment assigné à π_j était faisable sur π_j avant l'allocation de τ_i^0 (respectivement τ_i^1) et que la condition du Théoreme 10 est satisfaite, alors l'ensemble de tâches sur π_j reste ordonnançable après l'ajout de τ_i^0 (respectivement τ_i^1).□

Algorithme 1 Algorithme de partitionnement KTS-FF

Entrées : $m, k, maxK, \tau = \{\tau_1, ..., \tau_n\}$

Sorties : SUCCES et $\tau(\pi)$ si τ est ordonnançable, ECHEC sinon.

Fonction : KTS

```
début
pour i = 1 → |τ| faire
    Ordo ← faux ;
    /*Attribution de la tâche τ_i selon FFD*/
    pour j = 1 → m faire
        si τ_i ordonnançable sur π_j alors
            /* τ_i est attribuée à π_j */
            τ(π) = τ(π) ∪ τ_i ;
            Ordo ← vrai ;
            break ;
        fin si
    fin pour
    si not Ordo alors
        /* Essai de fractionnement de la tâche */
        si k < maxK alors
            /* La profondeur limite de fractionnement maxK n'est pas atteinte*/
            τ_i(O_i, C_i, T_i, D_i) est fractionnée en un ensemble composé de deux sous-tâches τ_i^0 et τ_i^1 :
            τ_i^split = {τ_i^0(O_i, C_i, 2 × T_i, D_i); τ_i^1(O_i + T_i, C_i, 2 × T_i, D_i)} ;
            si KTS(m, k + 1, maxK, τ_i^split) == SUCCES alors
                Ordo ← vrai ;
            fin si
        sinon
            /* La profondeur limite de fractionnement maxK a été atteinte*/
            Ordo ← faux ;
        fin si
    fin si
    si not Ordo alors
        retourner ECHEC ;
    fin si
fin pour
retourner SUCCES ;
fin
```

FIGURE 2.4 – Pseudo-code de KTS-FF

Le fonctionnement de KTS détermine, par des applications répétées du Lemme 2.1 au niveau de chaque tâche à assigner, l'ordonnançabilité de l'ensemble de tâches.

Théorème 17 *Considérant un ensemble de tâches périodiques τ, si l'algorithme de partitionnement KTS retourne SUCCES, alors le partitionnement établi est ordonnançable.*

Preuve: On observe que l'algorithme retourne SUCCES si et seulement s'il a assigné avec succès chaque tâche de τ aux différents processeurs. Avant l'assignation de la tâche τ_i, l'ensemble de tâches de chaque processeur est trivialement ordonnançable. Il résulte du Lemme 2.1 que tous les ensembles de tâches des processeurs restent ordonnançables après chaque attribution de tâche. Par conséquent, tous les ensembles de tâches des processeurs sont ordonnançables une fois que toutes les tâches de τ ont été assignées.□

3.3 Analyse de KTS

Nous cherchons à présent à établir quelle est la profondeur limite de l'algorithme KTS telle que le nombre de sous-tâches obtenues soit égal au nombre de processeurs de la plateforme. Cependant, il devient inutile de fractionner une tâche à partir du moment où son facteur d'utilisation individuel devient inférieur à la capacité inutilisée de tous les processeurs.

Lemme 2.2 *Au pire-cas, la profondeur maximale $maxK$ pouvant être atteinte par l'algorithme KTS pour fractionner une tâche τ_s en m sous-tâches est telle que $maxK = m - 1$.*

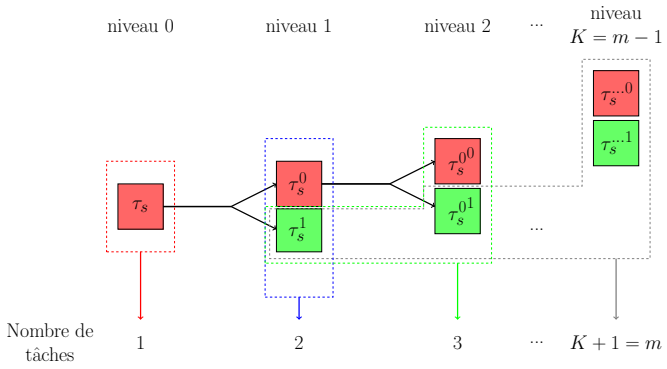


FIGURE 2.5 – Cas 1 : Profondeur maximale atteinte au pire-cas pour le fractionnement de τ_s en m sous-tâches

Preuve: La Figure 2.5 illustre le pire scénario, pour avoir m sous tâches dérivantes d'une tâche τ_s sur une profondeur maximale $maxK$. Si sur chaque niveau, une des tâches fractionnées est rejetée cela nous conduit à avoir au niveau K une nombre de tâches dérivantes égal à $2+K-1$, soit $K+1$. Comme nous souhaitons avoir m tâches dérivantes au maximum alors la profondeur maximale $maxK$ doit être égale à $m-1$ ($m = 2 + maxK - 1$).□

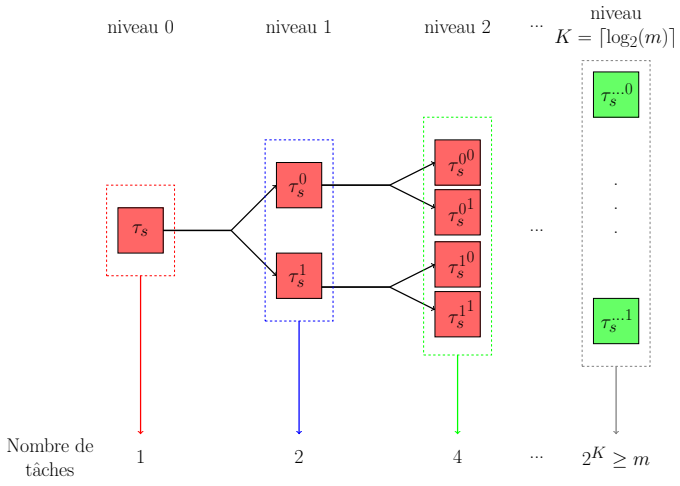


FIGURE 2.6 – Cas 2 : Profondeur minimale pour le fractionnement de τ_s en m sous-tâches

Lemme 2.3 *La profondeur minimale $minK$ nécessaire pour fractionner une tâche τ_s en m sous-tâches est telle que : $minK = \lceil log_2(m) \rceil$.*

Preuve: Pour avoir m sous-tâches dérivantes d'une tâche τ_s sur une profondeur minimale, il faut que sur chaque niveau la tâche soit fractionnée en deux autres tâches. Cela nous conduit à avoir au niveau K un nombre de tâches dérivantes égal à 2^K. Comme nous souhaitons avoir m

tâches dérivantes au maximum alors la profondeur minimale $minK$ doit être égale à $\lceil log_2(m) \rceil$ □.

Théorème 18 *La complexité de KTS est en* $O(n^2)$.

Preuve: Considérons un ensemble de n tâches à ordonnancer sur une plateforme à m processeurs :

- Si $n \leq m$, il n'y a pas de task splitting comme dans le pire des cas chaque tâche est assignée à un processeur.
  - avec KTS-FF : la première tâche passe le test sur le premier processeur et y est assignée (1 opération), la seconde passe son test au pire cas sur le premier et le second processeur (2 opérations), ..., la n-ième tâche passe le test au pire-cas sur les n processeurs avant d'être assignée (n opérations). Ce qui nous amène à dire que le nombre d'opérations total est de $1 + ... + n$ soit $n(n+1)/2$.
  - avec KTS-WF : chaque tâche passe le test d'ordonnançabilité sur les m processeurs avant de choisir celui qui peut la recevoir tout en maximisant sa capacité inutilisée. Le nombre d'opérations total est donc égal à $n.m$.
- Si $n > m$:
  - avec KTS-FF : les m premières tâches passent leur test comme décrit dans le cas de $n \leq m$ ($1 + 2 + ... + m = (m+1)m/2$ opérations) et sont toutes attribuées avec succès. Le reste de tâches $(n - m)$ passe au pire cas son test à chaque fois sur les m processeurs ($(n-m)m$ opérations). Si ces dernières n'arrivent pas à être assignées, elles sont splittées à chaque niveau k et donc elles doublent de nombre avant de re-tester à nouveau sur les m processeurs ($(n-m)m.2 + (n-m)m.2^2 + ... + (n-m)m.2^K$ opérations). Le nombre total d'opérations est donc égal à $m(m+1)/2 + (n-m)m \times K_s$ avec $K_s = \sum_{k=0}^{K} 2^k$ ayant une valeur constante.
  - avec KTS-WF : les m premières tâches passent leur test chacune sur les m processeurs comme décrit dans le cas de $n \leq m$ (m^2 opérations) et sont toutes attribuées avec succès. Le reste de tâches $(n - m)$ passe au pire cas son test à chaque fois sur les m processeurs ($(n-m)m$ opérations). Si ces dernières n'arrivent pas à être assignées, elles sont splittées à chaque niveau k et donc elles doublent de nombre avant de re-tester à nouveau sur les m processeurs ($(n-m)m.2 + (n-m)m.2^2 + ... + (n-m)m.2^K$ opérations). Le nombre total d'opérations est donc égal à $m^2 + (n-m)m \times K_s$ avec $K_s = \sum_{k=0}^{K} 2^k$ ayant une valeur constante.

Comme $n = \beta m$ avec $\beta \epsilon \mathbb{R}^+$, la complexité de KTS se trouve être en $O(n^2)$.

4 Validation par simulation

L'objectif de ce paragraphe est de décrire les simulations effectuées dans le but d'évaluer comparativement la performance de KTS vis-à-vis non seulement des heuristiques classiques de partitionnement mais également vis-à-vis de la stratégie EDF Split (DD) précédemment décrite. L'objectif est toujours de maximiser la proportion d'ensembles de tâches générés ordonnançables vis-à-vis du nombre total d'ensembles générés. Les expérimentations en simulation tendent également à évaluer l'impact du nombre de tâches, du nombre de processeurs et de la limite de profondeur K sur les performances relatives des différentes stratégies.

4.1 Environnement de simulation

L'environnement de simulation consiste en m ordonnanceurs locaux associés aux différents processeurs constituant la plateforme. Le programme de simulation a été implémenté en langage Perl. L'architecture fonctionnelle du simulateur est représentée sur la Figure 2.7.

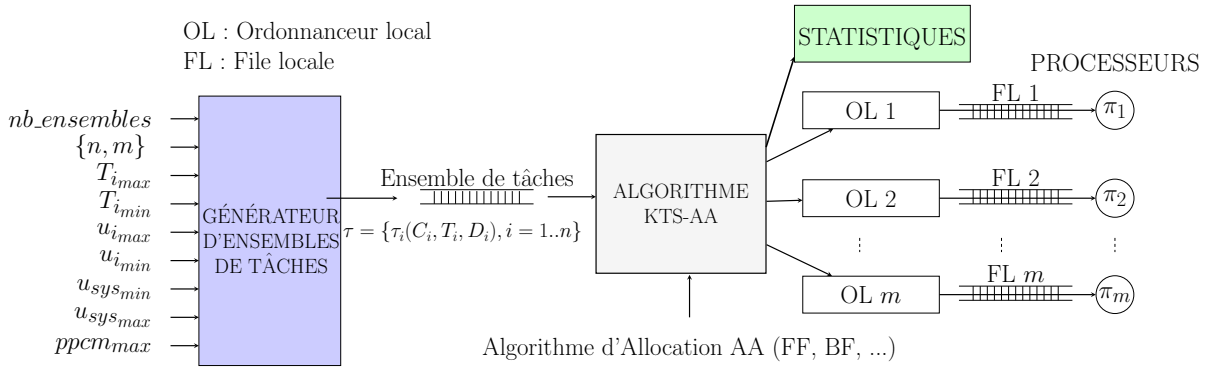


FIGURE 2.7 – Architecture fonctionnelle du simulateur

Le générateur d'ensembles de tâches périodiques a été conçu de manière à ce que les facteurs d'utilisation u_i des tâches de l'ensemble généré soient répartis de manière uniforme dans $[u_{i_{min}}, u_{i_{max}}] = [0.1, 1]$. Il accepte en entrée plusieurs paramètres : le nombre de tâches n souhaité pour la configuration, le ppcm maximum $ppcm_{max}$ des périodes des tâches choisies dans l'intervalle $[T_{i_{min}}, T_{i_{max}}] = [10, 200]$. La charge périodique totale du système u_{sys} (composé de m processeurs) varie dans $[u_{sys_{min}}, u_{sys_{max}}]$.
En sortie, on obtient pour chaque valeur de u_{sys}, $nb\_ensembles$ ensembles de tâches périodiques $\tau = \{\tau_i(C_i, T_i, D_i), i = 1..n\}$. Les durées d'exécution des tâches sont générées à partir des facteurs d'utilisation et des périodes affectés aux tâches, soit $C_i = u_i \times T_i$.

Pour chaque ensemble de tâches généré, le simulateur détermine s'il existe un partitionnement ordonnançable et ce, pour chaque valeur de u_{sys}, et calcule à la fin la proportion d'ensembles de tâches ordonnançables appelée le *taux de réussite* (*success ratio* en anglais). Le taux de réussite est le critère de performance qui permettra de comparer les différentes stratégies entre elles. Il est défini comme suit :

Définition 2.2 *Le taux de réussite (*success ratio *en anglais) représente le rapport entre le nombre d'ensembles de tâches jugés ordonnançables sur le nombre total d'ensembles de tâches générés, soit :*

$$taux\ de\ réussite = \frac{nb\_ensembles\_ordonnançable}{nb\_ensembles} \tag{2.6}$$

Dans notre méthode de génération de tâches, nous considérons deux séries d'expériences. La première concerne des tâches à échéances sur requêtes. La seconde est relative à des tâches à échéances contraintes (D_i est alors choisi aléatoirement dans $[C_i, T_i]$).

4.2 Résultats de simulation

Notre algorithme est comparé à l'heuristique de partitionnement classique FFDD et à l'algorithme EDF Split (DD). KTS-FFDD ($K = x$) correspond à l'algorithme KTS utilisé avec l'heuristique de partitionnement FFDD et possédant une profondeur limite de task splitting $K = x$.

Dans toutes nos simulations, nous générons 100 ensembles de tâches avec un facteur d'utilisation du système u_{sys} variant dans l'intervalle $[0.6, 1]$. Nos observations se font à chaque fois sur des ensembles composés de tâches (i) à échéances sur requêtes ($D = T$), (ii) à échéances contraintes ($D \leq T$).

4.2.1 Expérience 1 - Variation du nombre de tâches

Cas $D = T$:

La Figure 2.8 montre les résultats de performance pour des ensembles de tâches à échéances sur requêtes en faisant varier le nombre de tâches.

Nous pouvons noter l'augmentation des performances d'EDF Split (DD), KTS-FFDD et FFDD à mesure que le nombre de tâches dans les ensembles à partitionner augmente. Cela est dû au fait que les facteurs d'utilisation des tâches de l'ensemble (pour un facteur d'utilisation total u_{sys} donné) deviennent plus petits. Les tâches peuvent donc être plus facilement attribuées aux processeurs.

Les résultats indiquent clairement que les deux algorithmes KTS-FFDD (K = 2) et EDF Split (DD) sont meilleurs que l'heuristique de base FFDD. Ceci était prévisible car ces deux algorithmes essaient de bénéficier de la capacité inutilisée d'un processeur π_j chacun au travers de sa propre méthode de fractionnement des tâches.

Aussi, lorsque la profondeur K augmente, les performances de KTS augmentent et peuvent même dépasser celle de EDF Split (DD). En particulier, lorsque le système est fortement chargé (ex : u_{sys} = 0.925 et $n = m+1$ voir Figure 2.8(a)), FFDD arrive à ordonnancer seulement 26% des ensembles de tâches générés tandis que 60% le sont en utilisant KTS-FFDD avec K = 1, 80% en utilisant EDF Split (DD) et jusqu'à 93% en utilisant KTS-FFDD avec K = 2. Il est intéressant de noter qu'à travers des étapes successives de fractionnement, les tâches dérivantes voient leur facteur d'utilisation individuel diminuer et comme leurs échéances relatives ne changent pas, les tâches sont plus facilement assignées.

Les résultats montrent aussi qu'à mesure que le nombre de tâches augmente, la performance de KTS-FFDD (dès K = 1) devient similaire à celle d'EDF Split (DD), mais sans les coûts de migrations induits par ce dernier.

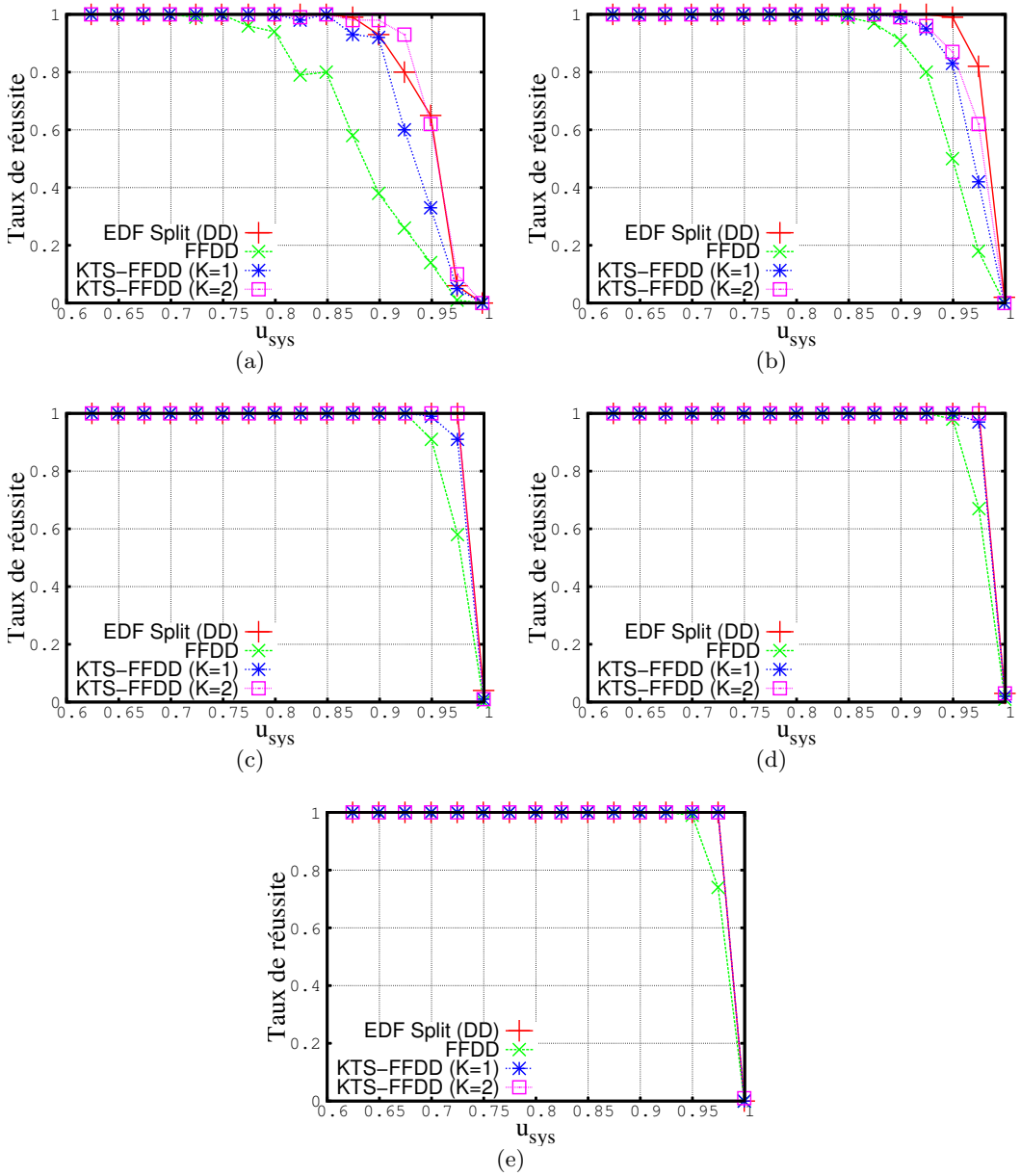


FIGURE 2.8 – Comparaison des performances de EDF Split (DD), FFDD et KTS-FFDD avec $D = T$ sur une plateforme à 4-processeurs pour des ensembles comportant :(a) $n = m + 1$, (b) $n = 2 \times m$, (c) $n = 3 \times m$, (d) $n = 4 \times m$, (e) $n = 5 \times m$ tâches.

Cas $D \leq T$:

La Figure 2.9 illustre les performances des différents algorithmes mais cette fois pour des ensembles de tâches à échéances contraintes. Comparé au cas où les tâches sont à échéances sur requêtes, nous notons que le taux de réussite observé est inférieur dans tous les cas à ceux de la Figure 2.8.

De plus, les résultats témoignent encore une fois que les performances des deux algorithmes KTS-FFDD (K = 2) et EDF Split (DD) dominent celles associées à l'heuristique FFDD. Néanmoins, le gain par rapport à FFDD dans le cas $D \leq T$ est plus petit. Par exemple, pour $u_{sys} = 0.925$ et $n = m + 1$ (voir Figure 2.9(a)), FFDD arrive à ordonnancer seulement 11% des ensembles de tâches générés tandis que 14% le sont en utilisant KTS-FFDD avec $K = 1$, 16% en utilisant EDF Split (DD) et jusqu'à 17% en utilisant KTS-FFDD avec $K = 2$.

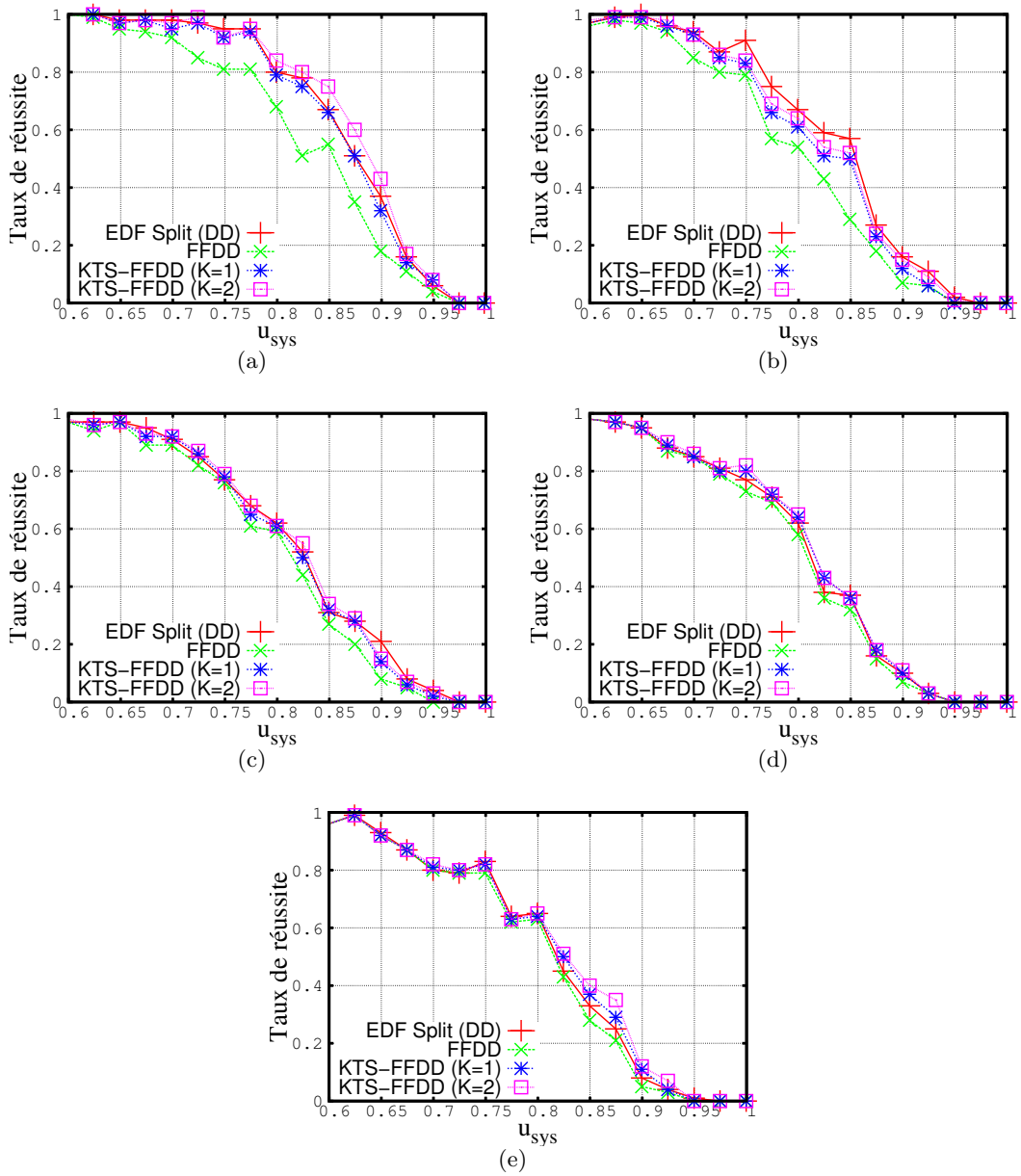


FIGURE 2.9 – Comparaison des performances de EDF Split (DD), FFDD et KTS-FFDD avec $D \leq T$ sur une plateforme à 4-processeurs pour des ensembles comportant :(a) $n = m + 1$, (b) $n = 2 \times m$, (c) $n = 3 \times m$, (d) $n = 4 \times m$, (b) $n = 5 \times m$ tâches.

4.2.2 Expérience 2 - Effet de $u_{i_{min}}$ et $u_{i_{max}}$

Dans cette expérience, il est question de voir l'effet du facteur d'utilisation minimum $u_{i_{min}}$ et $u_{i_{max}}$ sur l'efficacité des différents algorithmes sur une plateforme à 4-processeurs. En fonction du choix de $[u_{i_{min}}, u_{i_{max}}]$, nous pouvons considérer deux types de tâche dans l'ensemble : des tâches lourdes (ayant $u_i \geq 0.5$) et des tâches légères (ayant $u_i < 0.5$) dans l'ensemble.

Pour cela, nous nous plaçons dans trois cas :

1. $u_i \in [0.1, 0.5]$, toutes les tâches sont légères.
2. $u_i \in [0.1, 1]$, il y a autant de tâches légères que de tâches lourdes.
3. $u_i \in [0.5, 1]$, toutes les tâches sont lourdes.

Les Figures 2.10 et 2.11 illustrent les performances des différents algorithmes sur une plateforme à 4-processeurs dans les trois cas énoncés précédemment pour respectivement des tâches

à échéances sur requêtes et des tâches à échéances contraintes.

Les résultats indiquent clairement que les deux algorithmes KTS-FFDD (K = 2) et EDF Split (DD) sont meilleurs que l'heuristique de base FFDD et ce quel que soit le cas considéré. De plus, nous remarquons que plus le taux de tâches lourdes augmente, plus EDF Split et KTS-FFDD sont efficaces face à FFDD. Pour en témoigner,

1. lorsque $D = T$ (voir Figure 2.10) et que l'utilisation du système u_{sys} = 0.925, parmi tous les ensembles de tâches générés,
   - KTS-FFDD avec K = 2 en ordonnance (i) 2% de plus lorsque $u_i\epsilon$ [0.1,0.5], (ii) 67% de plus lorsque $u_i\epsilon$ [0.1,1] et (iii) 78% de plus lorsque $u_i\epsilon$ [0.5,1].
   - EDF Split (DD) en ordonnance (i) 2% de plus lorsque $u_i\epsilon$ [0.1,0.5], (ii) 54% de plus lorsque $u_i\epsilon$ [0.1,1] et (iii) 65% de plus lorsque $u_i\epsilon$ [0.5,1].

2. lorsque $D \leq T$ (voir Figure 2.11) et que l'utilisation du système u_{sys} = 0.675, parmi tous les ensembles de tâches générés,
   - KTS-FFDD avec K = 2 en ordonnance (i) 1% de plus lorsque $u_i\epsilon$ [0.1,0.5], (ii) 4% de plus lorsque $u_i\epsilon$ [0.1,1] et (iii) 59% de plus lorsque $u_i\epsilon$ [0.5,1].
   - EDF Split (DD) en ordonnance (i) 0% de plus lorsque $u_i\epsilon$ [0.1,0.5], (ii) 4% de plus lorsque $u_i\epsilon$ [0.1,1] et (iii) 69% de plus lorsque $u_i\epsilon$ [0.5,1].

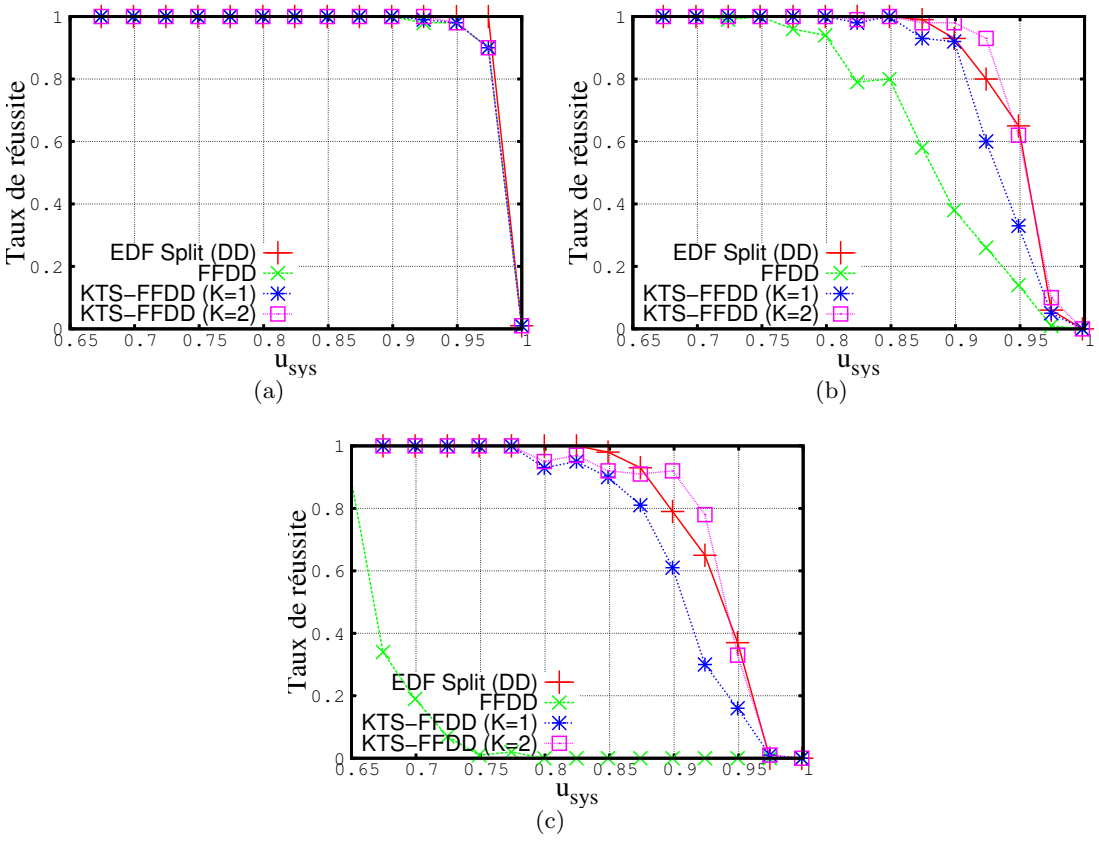


FIGURE 2.10 – Comparaison des performances de EDF Split (DD), FFDD et KTS-FFDD avec $D = T$ sur une plateforme à 4-processeurs avec $u_i\epsilon$:(a) [0.1,0.5], (b) [0.1,1], (c) [0.5,1]

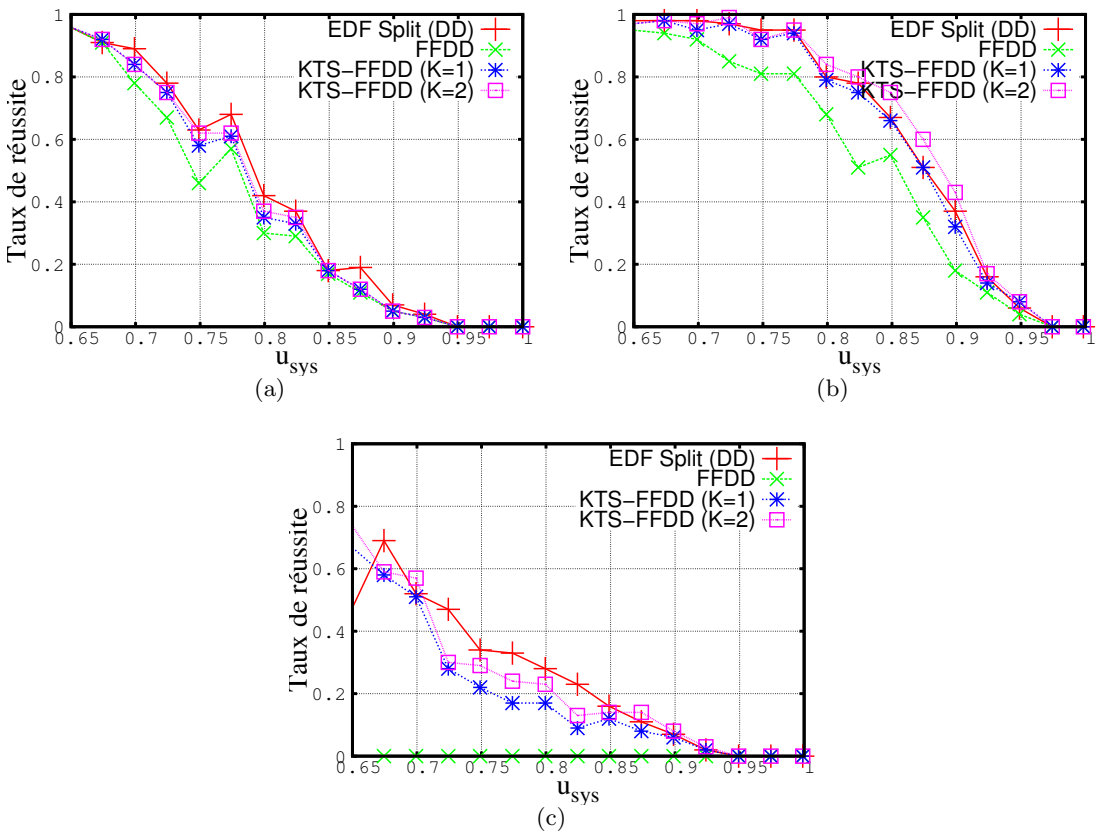


FIGURE 2.11 – Comparaison des performances de EDF Split (DD), FFDD et KTS-FFDD avec $D \leq T$ sur une plateforme à 4-processeurs avec $u_i \epsilon$:(a) [0.1,0.5], (b) [0.1,1], (c) [0.5,1]

En conclusion, lorsque les tâches sont à échéances sur requêtes KTS-FFDD apparaît d'autant plus comme une alternative intéressante vis-à-vis de EDF Split (DD) que l'ensemble comporte majoritairement des tâches lourdes. Dans le cas de tâches à échéances contraintes, KTS-FFDD offre de meilleures performances que EDF Split (DD) pour des ensembles de tâches mixtes comportant autant de tâches légères que de tâches lourdes.

4.2.3 Expérience 4 - Variation du nombre de processeurs

Les Figures 2.12 et 2.13 illustrent les performances des différents algorithmes en faisant varier le nombre de processeurs m dans le cas de tâches à échéances sur requêtes et de tâches à échéances contraintes respectivement.

Nous notons premièrement qu'à mesure que l'on augmente le nombre de processeurs dans le système, les performances de KTS-FFDD et EDF Split (DD) augmentent. De plus, le taux de réussite observé pour KTS-FFDD et EDF Split (DD) dépasse clairement celui de FFDD dans tous les cas observés.

KTS-FFDD devient d'autant plus efficace vis-à-vis de EDF Split (DD) que le nombre de processeurs composant la plateforme augmente. En particulier,

1. lorsque $D = T$ (voir Figure 2.12) et que l'utilisation du système $u_{sys} = 0.975$,
   - $m = 4$: FFDD arrive à ordonnancer 18% des ensembles de tâches générés tandis que 82% des ensembles de tâches sont ordonnançables en utilisant EDF Split (DD) et 62% le sont en utilisant KTS-FFDD avec $K = 2$.
   - $m = 8$: FFDD arrive à ordonnancer 9% des ensembles de tâches générés tandis que 82% des ensembles de tâches sont ordonnançables en utilisant EDF Split (DD) et 90% le sont en utilisant KTS-FFDD avec $K = 3$.

– $m = 16$: FFDD arrive à ordonnancer 19% des ensembles de tâches générés tandis que 96% des ensembles de tâches sont ordonnançables en utilisant EDF Split (DD) et 99% le sont en utilisant KTS-FFDD avec $K = 4$.

2. lorsque $D \leq T$ (voir Figure 2.13) et que l'utilisation du système $u_{sys} = 0.875$,

– $m = 4$: FFDD arrive à ordonnancer 18% des ensembles de tâches générés tandis que 27% des ensembles de tâches sont ordonnançables en utilisant EDF Split (DD) et 24% le sont en utilisant KTS-FFDD avec $K = 2$.
– $m = 8$: FFDD arrive à ordonnancer 9% des ensembles de tâches générés tandis que 17% des ensembles de tâches sont ordonnançables en utilisant EDF Split (DD) et 24% le sont en utilisant KTS-FFDD avec $K = 3$.
– $m = 16$: FFDD arrive à ordonnancer 16% des ensembles de tâches générés tandis que 38% des ensembles de tâches sont ordonnançables en utilisant EDF Split (DD) et 45% le sont en utilisant KTS-FFDD avec $K = 4$.

Les très bonnes performances de l'algorithme KTS s'expliquent par le fait qu'une tâche peut être fractionnée en au plus m nouvelles tâches qui vont être attribuées aux m processeurs. Par conséquent, plus on augmente le nombre de processeurs, plus on autorise la tâche à se fractionner en un nombre important de sous tâches dont le facteur d'utilisation devient de plus en plus petit, permettant par là-même de les assigner plus facilement à des processeurs.

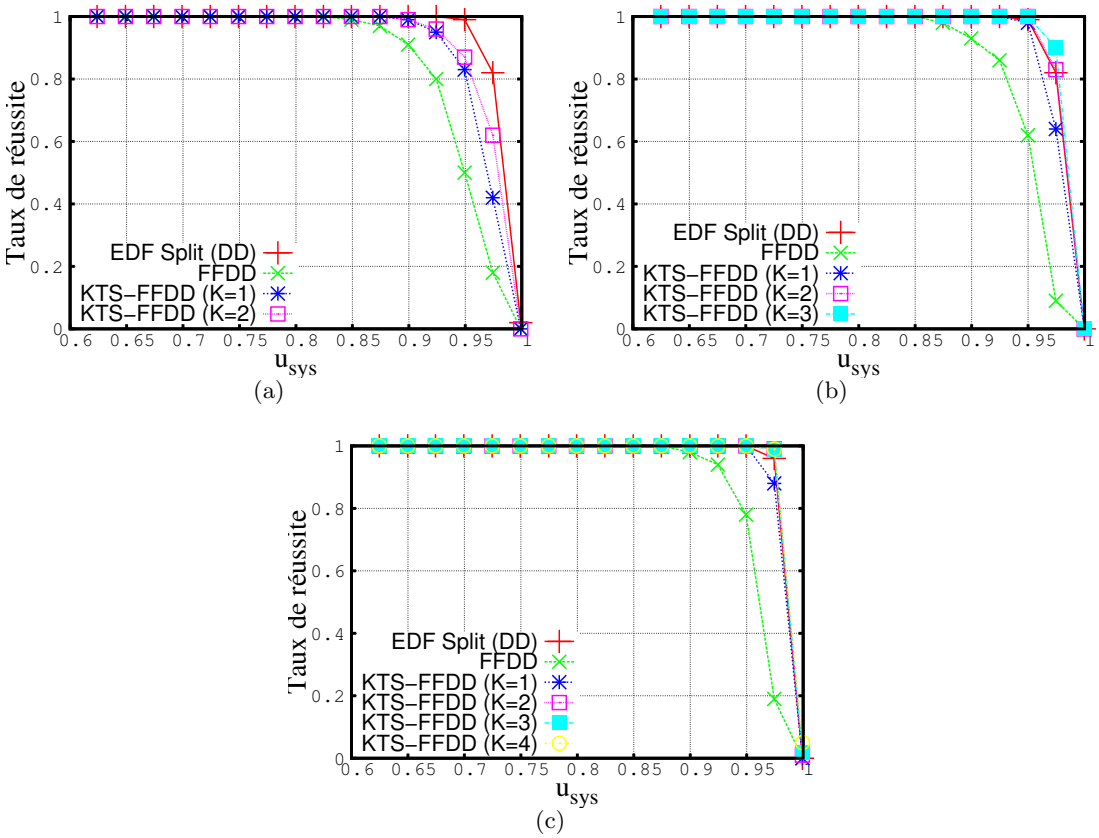


FIGURE 2.12 – Comparaison des performances de EDF Split (DD), FFDD et KTS-FFDD avec $D = T$ et $n = 2 \times m$ sur différentes plateformes à m-processeurs :(a) $m = 4$, (b) $m = 8$, (c) $m = 16$

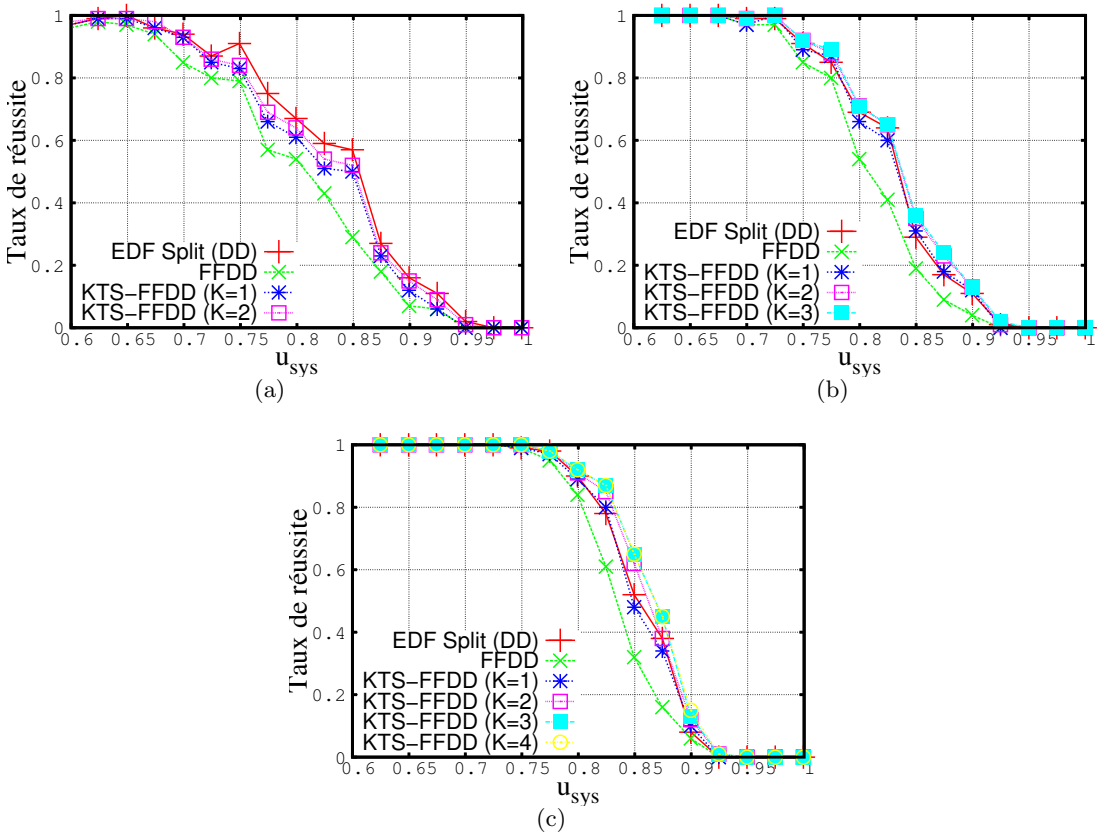


FIGURE 2.13 – Comparaison des performances de EDF Split (DD), FFDD et KTS-FFDD avec $D \leq T$ et $n = 2 \times m$ sur différentes plateformes à m-processeurs :(a) $m = 4$, (b) $m = 8$, (c) $m = 16$

5 Conclusion

Cette section a introduit un nouvel algorithme de partitionnement temps réel basé sur le fractionnement de tâches sur une plateforme multiprocesseur. L'approche KTS est conceptuellement simple, facile à mettre en œuvre dans les systèmes existants, et efficace. Chaque tâche fractionnée reste une tâche périodique dont le code peut être dupliqué, sans impliquer de quelconques migrations.

Au travers de ce travail, nous avons montré comment l'analyse classique d'ordonnançabilité selon EDF placée dans le contexte de tâches périodiques asynchrones peut être utilisée dans un tel algorithme de fractionnement de tâches.

L'évaluation sur une large gamme d'ensembles de tâches, générés de manière aléatoire, indique que l'algorithme KTS fournit un bon compromis en offrant des performances intermédiaires entre l'approche semi-partitionnée existante EDF split (DD) et l'heuristique de partitionnement classique FFDD. Il faut cependant noter que l'ordonnançabilité compétitive de KTS est obtenue sans aucun changement de contexte supplémentaire et sans aucun coût de migration et ce, quel que soit le type de la tâche ($D = T$ ou $D \leq T$, lourde/légère) et le nombre de processeurs du système. Toutefois, KTS aura tendance à être plus efficace par rapport à FFDD sur des plateformes où le nombre de processeurs est plus important et pour des ensembles composés à majorité de tâches lourdes.

KTS, comparé à EDF Split (DD), ne peut gérer le cas de tâches sporadiques. Mais, il peut être mis en œuvre dans de nombreuses applications tandis qu'EDF Split (DD) est utilisé uniquement dans des systèmes dans lesquels un job de tâche (un bloc d'instructions) peut être divisé en deux parties assignées sur deux processeurs différents ; ce qui n'est pas toujours le cas.

Le Tableau 2.2 fournit une évaluation qualitative des algorithmes présentés dans ce chapitre en termes de performance, de complexité calculatoire, de complexité d'implémentation et d'overhead temporel :

Algorithmes	Performance	Complexité calculatoire (hors-ligne)	Complexité d'implémentation	Overhead temporel (en-ligne)
FFDD	☹	☺	☺	☺
EDF Split DD	☺	☹	😐	😐
KTS-FFDD ($K = x$)	☺	☹	☺	☺

TABLE 2.2 – Synthèse

L'objet du chapitre suivant portera sur l'extension de cette approche à la catégorie de tâches à contraintes fermes permettant à certains jobs d'être ignorés ou abandonnés au cours de leur exécution sans conséquences graves sur le système, notamment pour faire face à des cas de surcharge temporaire de traitement.

Chapitre 3

Contribution au partitionnement sous contraintes de QoS

Ce chapitre est consacré au problème du partitionnement étendu à un modèle de tâches temps réel à contraintes fermes sur une plateforme multiprocesseur homogène. Nous présentons une formalisation du problème et décrivons les conditions d'ordonnançabilité exactes pour le modèle de tâches et la plateforme considérés. Nous adaptons ensuite les heuristiques de partitionnement classiques au modèle de tâches temps réel fermes Skip-Over (modèle dans lequel certains jobs de tâche peuvent être ignorés ou abandonnés sans conduire à la faute du système). Puis, dans un deuxième temps, nous proposons une extension de l'algorithme KTS (décrit dans le chapitre 2) à un fonctionnement tolérant aux fautes. La performance des stratégies développées est validée au travers de différentes simulations dans lesquelles nous explorons, en particulier, l'effet du facteur de pertes autorisées au niveau des tâches sur la proportion d'ensembles ordonnançables.

1 Modèle du système

Nous considérons une plateforme π composée de m processeurs identiques : $\pi = \{\pi_1, ..., \pi_m\}$, et un ensemble de n tâches indépendantes, périodiques et préemptibles $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Chaque tâche $\tau_i\,(O_i, C_i, T_i, D_i, s_i)$ est définie par :

- O_i, l'offset de τ_i ;
- C_i, son temps d'exécution pire-cas ;
- T_i, sa période ;
- D_i, son délai critique ou échéance relative, avec $D_i \leq T_i$;
- s_i, le paramètre de pertes de τ_i conformément au modèle Skip-Over [KS95].

Le modèle considéré est dit profondément rouge ou *deeply-red* en anglais, c'est-à-dire que les tâches sont activées de manière synchrone et que les $s_i - 1$ premiers jobs de chaque tâche sont rouges.

Chaque tâche est caractérisée par : (i) un facteur d'utilisation : $u_i = \frac{C_i}{T_i}$, (ii) un facteur d'utilisation équivalent intégrant les pertes autorisées : $u_i^* = \frac{C_i}{T_i} \times \frac{s_i - 1}{s_i}$, (iii) une densité : $\delta_i = \frac{C_i}{min\{D_i, T_i\}}$, et (iv) une densité équivalente intégrant les pertes autorisées : $\delta_i^* = \frac{C_i}{min\{D_i, T_i\}} \times \frac{s_i - 1}{s_i}$. Le facteur d'utilisation équivalent de l'ensemble de tâches Γ_j assignées à π_j est noté $U_p^* = \sum_{\tau_i \epsilon \Gamma_j} u_i^*$.

Chaque processeur π_j doit assurer, au pire cas, l'exécution des jobs rouges des tâches du sous-ensemble Γ_j. Nous noterons $H(\Gamma_j) = PPCM(T_1, ..., T_n)$ l'hyperpériode de Γ_j et $H^*(\Gamma_j) = PPCM(s_1 T_1, ..., s_n T_n)$ l'hyperpériode équivalente de Γ_j (considérant les pertes autorisées). Les surcoûts en temps, occasionnés par les éventuelles préemptions, sont supposés nuls.

2 Heuristiques de partitionnement adaptées à des systèmes tolérants aux pertes

Les heuristiques de partitionnement classiques adaptées à des systèmes tolérants aux pertes (sous contraintes de QoS) sont dénotées FF_{QoS} (First Fit for system under QoS constraints), BF_{QoS} (Best Fit for system under QoS constraints), et WF_{QoS} (Worst Fit for system under QoS constraints).

2.1 Principe du partitionnement sous contraintes de Qualité de Service

Le partitionnement consiste à assigner les tâches aux différents processeurs selon une heuristique donnée. Toutefois, sous contraintes de QoS, l'algorithme de partitionnement dispose d'une flexibilité accrue apportée par le modèle de tâches tolérant aux fautes. En effet, le partitionnement est plus aisé dans la mesure où l'algorithme atteint moins rapidement le point pour lequel l'ajout d'une nouvelle tâche à un processeur donné conduit à la non-ordonnançabilité de l'ensemble de tâches (voir Figure 3.1). Ainsi, sur chaque processeur, nous veillons de manière *hors – ligne* à ce que la charge équivalente de chaque processeur π_j (c-à-d intégrant les pertes autorisées) ne dépasse pas 100%.

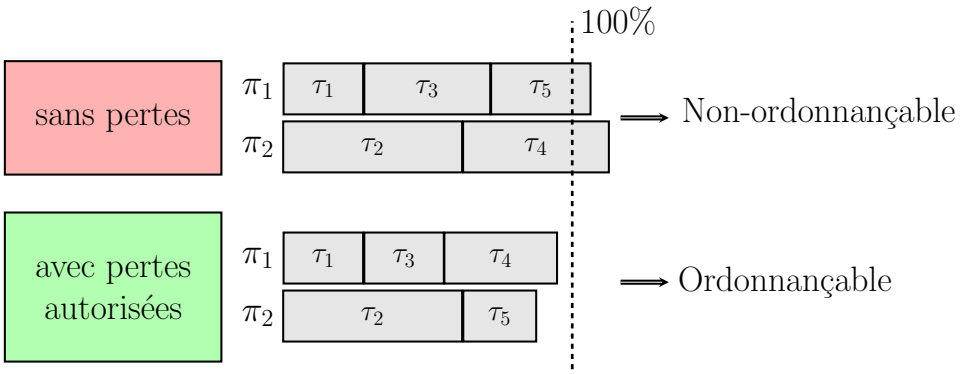


FIGURE 3.1 – Flexibilité induite grâce aux pertes

Comme le montre la Figure 3.1, si les tâches τ_3, τ_4 et τ_5 tolérent par exemple de ne pas exécuter un job sur trois, leur facteur d'utilisation se trouve réduit de 33.33%, ce qui permet alors de partitionner l'ensemble de tâches sur π_1 et π_2.

2.2 Formulation du problème

Le problème soulevé ici consiste à trouver un partitionnement *valide* c'est-à-dire satisfaisant les contraintes temporelles des jobs rouges uniquement. Nous utilisons l'algorithme Red Tasks Only (RTO) [KS95] avec lequel les jobs rouges des tâches sont ordonnancés au plus tôt selon l'algorithme EDF, tandis que les jobs bleus sont systématiquement rejetés. Cet algorithme est optimal [KS95] considérant un modèle *profondément rouge* (deeply red). L'objectif est de déterminer la répartition d'un ensemble des tâches fermes $\tau = \{\tau_1, \tau_2, ...\tau_n\}$ sur les différents processeurs de la plateforme $\Pi = \{\pi_1, ..., \pi_m\}$, de manière à ce que la configuration globale soit ordonnançable. Plus formellement, il s'agit de partitionner l'ensemble des n tâches en m sous-ensembles disjoints $\Gamma_1, \Gamma_2, ..., \Gamma_m$ (avec $\bigcup \Gamma_j = \tau$) et d'ordonnancer ensuite chaque sous-ensemble Γ_j sur le processeur π_j avec une politique d'ordonnancement monoprocesseur EDF sachant que sur chaque processeur, tous les jobs rouges doivent s'exécuter dans le respect de leurs échéances.

Caccamo et Buttazzo ont prouvé dans [CB98] que considérant un modèle profondément rouge, un ensemble de tâches à échéances sur requêtes tolérantes aux pertes au sens Skip-Over est faisable sur un processeur si et seulement si son facteur d'utilisation équivalent du processeur

U_p^* est inférieur à 1 (Théorème 16 page 33). Nous généralisons cette condition en introduisant la notion de charge équivalente du processeur pour un ensemble de tâches synchrones à échéances contraintes. Relativement à un processeur π_j, et un ensemble de tâches τ, nous dénoterons cette quantité $Load_j^*(\tau)$.

Définition 3.1 *Étant donné un ensemble $\tau = \{\tau_i(C_i, T_i, D_i, s_i)\}$ de n tâches périodiques synchrones à échéances contraintes autorisant des pertes au sens Skip-Over et s'exécutant sur un processeur π_j, la charge équivalente du processeur est définie par :*

$$Load_j^*(\tau) = max_{L\in[D_{min},P)} \frac{DBF_{QoS}(\tau, L)}{L} \tag{3.1}$$

où $DBF_{QoS}(\tau, L)$ représente la demande processeur liée à l'exécution des jobs rouges de τ dans l'intervalle $[0, L]$ et qui, selon le modèle profondément rouge, est définie comme suit :

$$DBF_{QoS}(\tau, L) = \sum_{i=1}^{n} D(i, [0, L]) \tag{3.2}$$

avec

$$D(i, [0, L]) = \left(1 + \lfloor \frac{L - D_i}{T_i} \rfloor - \lfloor \frac{1}{s_i} \left(1 + \lfloor \frac{L - D_i}{T_i} \rfloor\right) \rfloor\right) \times C_i \tag{3.3}$$

$1 + \lfloor \frac{L-D_i}{T_i} \rfloor$ représente le nombre de jobs de τ_i actifs et ayant leur échéance dans $[0, L]$ tandis que $\lfloor \frac{1}{s_i} \left(1 + \lfloor \frac{L-D_i}{T_i} \rfloor\right) \rfloor$ représente le nombre de jobs bleus dans $[0, L]$. La différence des deux termes donne le nombre de jobs rouges qui doivent s'exécuter impérativement dans $[0, L]$. Ce nombre multiplié par C_i nous donne le temps d'exécution nécessaire pour l'exécution des jobs rouges dans $[0, L]$. Nous calculons DBF_{QoS} seulement pour des valeurs de L correspondant aux échéances absolues des jobs rouges sur $[0, H(\tau)]$.

Considérons à présent un ensemble de tâches périodiques synchrones à échéances contraintes. Nous énonçons alors le théorème suivant :

Théorème 19 *Un ensemble de tâches périodiques Γ_j à échéances contraintes tolérantes aux pertes au sens Skip-Over et profondément rouge est ordonnançable sur un processeur π_j selon RTO si et seulement si la charge du processeur équivalente à l'exécution des jobs rouges de tâches ne dépasse pas la capacité de π_j, à savoir :*

$$Load_j^*(\Gamma_j) \leq 1 \tag{3.4}$$

Preuve: Si. Par hypothèse (modèle deeply-red), les jobs de tâches se réveillent à l'instant 0 et sont rouges. Selon l'algorithme RTO associé au modèle Skip-Over, pour chaque tâche τ_i, chaque s_i-ème job est systématiquement ignoré (exactement les s_i-ème, $2 * s_i$-ème, $3 * s_i$-ème, ... jobs qui sont ignorés). Tous les autres jobs (jobs rouges) sont exécutés selon EDF. $D(i, [t_1, t_2])$ représente la demande de traitement de τ_i dans l'intervalle $[t_1, t_2]$ selon EDF, à savoir le temps total de traitement nécessité pour l'exécution de tous les jobs rouges de τ_i qui sont réveillés et doivent s'exécuter dans l'intervalle $[t_1, t_2]$. Supposons que la condition $Load_j^*(\tau) \leq 1$ est vraie et qu'à un certain moment t, un job manque son échéance. Soit $t_a \geq 0$ la dernière fois (avant t) où le processeur a été laissé inactif par l'algorithme d'ordonnancement ($t_a = 0$, s'il n'y en a pas). Soit $t_b \geq 0$ la dernière fois (avant t) où le processeur était occupé à exécuter un job de tâche possèdant une échéance après t ($t_b = 0$ s'il n'y en a pas). Soit $t' = \max(t_a, t_b)$. Le motif de pertes de l'algorithme assure que $D(i, [t', t]) \leq D(i, [0, t - t'])$. L'instant t' possède la propriété que seuls les jobs rouges de tâche qui ont été activés après t' avec une échéance absolue inférieure ou égale à t sont exécutés sur $[t', t]$. Il n'y a aucun moment dans $[t', t]$ où le processeur est inactif. Le fait que l'échéance ait été manquée signifie que la demande de traitement $C[t', t]$ au cours

de cet intervalle est supérieure à la longueur de l'intervalle . Autrement dit,

$$t - t' < C[t', t] \tag{3.5}$$

Ce qui nous conduit à,

$$t - t' < C[t', t] = \sum_{i=1}^{n} D(i, [t', t]) \leq DBF_{QoS}(\Gamma_j, [0, t - t']) \leq (t - t').Load_j^*(\Gamma_j) \tag{3.6}$$

et donc $Load_j^*(\Gamma_j) > 1$ qui nous amène à une contradiction.□
Seulement si. Supposons que $Load_j^*(\Gamma_j) > 1$ sur un intervalle $[0, t]$, la charge de traitement $C[0, t]$ nécessitée pour l'exécution des jobs rouges dans le pire cas est $C[0, t] = DBF_{QoS}(\Gamma_j, t)$. Selon la définition de $Load_j^*(\Gamma_j), \exists t$ tel que $Load_j^*(\Gamma_j).t = DBF_{QoS}(\Gamma_j, t)$. Par conséquent, $C[0, t] = DBF_{QoS}(\Gamma_j, t) = DBF_{QoS}(\Gamma_j, t) > t$. Comme le temps de traitement sur $[0, t]$ dépasse la longueur de ce même intervalle alors Γ_j n'est pas faisable. □

Il est suffisant de vérifier la condition du Théorème 19 pour les instants t correspondant aux échéances absolues des jobs. De plus, il est inutile de tester cette condition pour des instants supérieurs à $H(\tau)$ [KS95].

Considérons maintenant un ensemble de tâches fermes asynchrones où les $s_i - 1$ premiers jobs de chaque tâche sont rouges. Nous énonçons alors le théorème suivant :

Théorème 20 *Un ensemble $\tau = \{\tau_i(O_i, C_i, T_i, D_i, s_i)\}$ de n tâches périodiques asynchrones à échéances contraintes autorisant des pertes au sens Skip-Over est ordonnançable selon RTO sur un processeur π_j si et seulement si toutes les échéances des jobs rouges sont respectées sur l'intervalle $[0, 2 \times (max_{i=1}^{n}(O_i) + 2H^*(\tau))]$.*

Preuve:
Si. Pour chacune des tâches ordonnancées selon RTO, il y a un motif qui se répète dans la séquence d'ordonnancement générée : les $s_i - 1$ jobs s'exécutent puis ensuite un job est ignoré. Considérant ce motif comme une tâche τ_k de période $T_k = s_i T_i$, et comme l'ont démontré Leung et Merill dans [LM80], pour des tâches asynchrones périodiques, il existe une cyclicité de traitement. Cette fenêtre (pas nécessairement la plus courte) correspond pour τ_k à l'intervalle $[0, max_{i=1}^{n}(O_i) + 2 \times PPCM(T_k, i = 1..n)])$, soit également $[0, max_{i=1}^{n}(O_i) + 2H^*(\tau)])$. Cependant, si la fin de la fenêtre (cyclique) coïncide à l'intérieur du motif $(s_i T_i)$, le temps d'exécution d'un ou plusieurs jobs du motif se sera pas pris en compte dans le calcul de la charge processeur. Nous vérifions alors sur une fenêtre de plus pour considérer ces jobs, d'où l'intervalle $[0, 2 \times (max_{i=1}^{n}(O_i) + 2H^*(\tau))]$.
Seulement si. Il est clair que si au moins une échéance de job rouge n'est pas respectée, l'ensemble ne peut être ordonnançable.

3 Extension de KTS à des systèmes tolérants aux fautes

3.1 Fonctionnement

Jusqu'ici, l'approche KTS à un modèle de tâches à échéances strictes a permis d'augmenter de manière significative les performances par rapport aux heuristiques de partitionnement classiques. Le choix d'étendre cette approche aux systèmes tolérants aux fautes est dénotée $KTS - AA_{QoS}$ (AA : l'heuristique de partitionnement). Cette variante va permettre de diminuer le taux de rejet dû au non respect des contraintes temporelles grâce à la flexibilité apportée par le modèle de tâches fermes en plus de la méthode de task splitting.

Le principe de $KTS - AA_{QoS}$ est toujours le même. Toutefois, les tests d'ordonnnançabilité reposent, cette fois-ci, sur le seul respect des contraintes temporelles des jobs rouges. Les tâches

sont d'abord assignées aux différents processeurs selon une certaine heuristique de base (FFDU : First Fit Decreasing Utilization, BFDD : Best Fit Decreasing Density) basée, par exemple, sur l'utilisation ou la densité. Dès lors que l'allocation d'une tâche sur un processeur n'est plus possible (rejet dû à une violation d'échéance de job rouge), la tâche rejetée va subir un fractionnement comme décrit dans les Figures 3.2 et 3.3 et ce, en fonction de la valeur de son facteur de pertes.

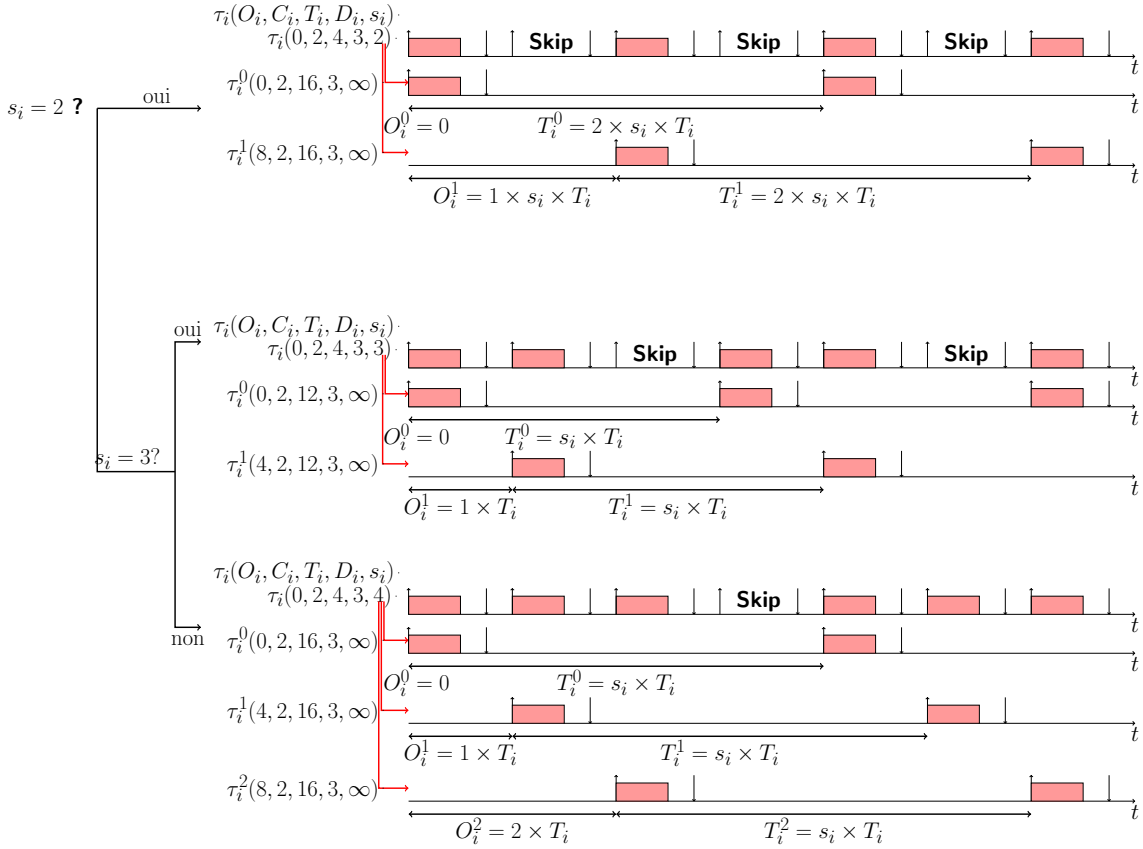


FIGURE 3.2 – KTS_{QoS} : Exemple du découpage d'une tâche τ_i tolérant les pertes sur 4 processeurs

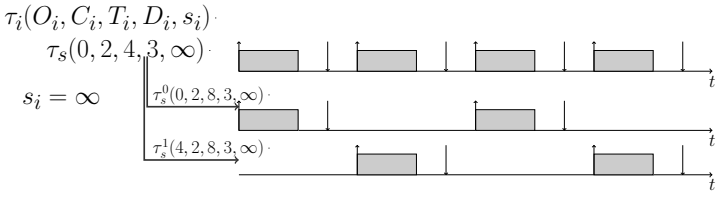


FIGURE 3.3 – KTS_{QoS} : Exemple du découpage d'une tâche τ_i ne tolérant aucunes pertes

Comme l'illustre la Figure 3.2, si une tâche temps réel tolérante aux fautes ne peut être assignée alors deux cas de figures se présentent ; si la tâche est :

1. temps réel ferme et a un taux de pertes s_i égal à 2, alors de cette tâche dérivent deux nouvelles tâches temps réel à échéances strictes τ_i^0 et τ_i^1 ayant les caractéristiques suivantes :
   - $\tau_i^0(O_i, C_i, 2 \times s_i \times T_i, D_i, \infty)$
   - $\tau_i^1(O_i + s_i \times T_i, C_i, 2 \times s_i \times T_i, D_i, \infty)$
2. temps réel ferme et a un taux de pertes s_i strictement supérieur à 2, alors de cette tâche dérive un ensemble composé de $s_i - 1$ nouvelles tâches à échéances strictes ayant les

propriétés suivantes : $\tau_i^{split} = \{\tau_i^x(O_i + x \times T_i, C_i, s_i \times T_i, D_i, \infty), x = 0..(s_i - 2)\}$

3. temps réel dur ($s_i = \infty$) alors le découpage de la tâche est le même que celui observé sous KTS : la tâche se découpe en un ensemble composé de 2 tâches τ_i^0 et τ_i^1 ayant les propriétés suivantes (voir Figure 3.3) :
   - $\tau_i^0(O_i, C_i, 2 * T_i, D_i, \infty)$
   - $\tau_i^1(O_i + T_i, C_i, 2 * T_i, D_i, \infty)$

Ensuite, chaque nouvelle tâche tente d'être assignée à un processeur. Si une tâche découpée ne peut être affectée à un processeur, celle-ci est découpée à nouveau. Notons qu'une tâche dérivant d'une tâche ferme devient stricte mais possède un facteur d'utilisation réduit tenant compte du facteur de perte. A chaque étape de découpage correspond un niveau. On définira une profondeur limite K, c'est-à-dire qu'une tâche ne pourra pas subir plus de K dérivations.

3.2 Description algorithmique

La Figure 3.4 montre le pseudo-code de l'algorithme KTS utilisé avec l'heuristique FF_{QoS}, et dénommé ci-après $KTS - FF_{QoS}$. k represente le niveau de *task splitting* courant atteint et $maxK$ est la profondeur limite de *task splitting* (noté précédemment par K). Sur une plateforme multiprocesseur, $\tau(\pi)$ désigne l'ensemble de tâches parmi τ_1,..., τ_{i-1} ayant été assignées avec succès à l'un des processeurs de l'ensemble π. Initialement, $\tau(\pi) = \varnothing$.

Au début, l'algorithme fonctionne selon une heuristique de partitionnement donnée. Si aucun processeur n'est capable de recevoir une tâche donnée, nous ne pouvons rien conclure quant à l'ordonnançabilité de l'ensemble de tâches τ sur la plateforme multiprocesseur. La tâche τ_i est alors fractionnée en un sous-ensemble τ_i^{split} composé de a tâches sachant que $a = 2$ si $s_i = \{2; \infty\}$, $a = s_i - 1$ sinon. Celles-ci sont assignées à un sous-ensemble de processeurs de la plateforme à m processeurs. Le Lemme 3.1 affirme qu'en attribuant une tâche fractionnée de τ_i^{split} à un processeur π_j, l'ordonnançabilité des tâches confiées précédemment à l'ensemble des processeurs reste intacte.

Lemme 3.1 *Si l'ensemble de tâches précédemment assignées aux processeurs est ordonnançable (sur chaque processeur) et que par la suite l'algorithme $KTS-AA_{QoS}$ assigne une tâche τ_i^0 (respectivement τ_i^1) au processeur π_j (d'après le Théoreme 20), alors l'ensemble de tâches assignées à chaque processeur (y compris processeur π_j) reste ordonnançable.*

Preuve: On observe que l'ordonnançabilité des processeurs autres que le processeur π_j n'est pas affectée par l'allocation de la tâche τ_i^0 (respectivement τ_i^1) au processeur π_j. Aussi, si l'ensemble de tâches précédemment assigné à π_j était ordonnançable sur π_j avant l'allocation de τ_i^0 (respectivement τ_i^1) du fait que tous leurs jobs rouges s'exécutent avant leurs échéances et que la condition du Théoreme 20 est satisfaite, alors l'ensemble de tâches sur π_j reste ordonnançable après l'ajout de τ_i^0 (respectivement τ_i^1).□

Algorithme 2 Algorithme de partitionnement $KTS - FF_{QoS}$

Entrées : $m, k, maxK, \tau = \{\tau_1, ..., \tau_n\}$

Sorties : SUCCES et $\tau(\pi)$ si τ est ordonnançable, ECHEC sinon.

Fonction : $KTS - FF_{QoS}$

début
pour $i = 1 \rightarrow |\tau|$ **faire**
  $Ordo \leftarrow faux$;
  /*Attribution de la tâche τ_i selon FFD_{QoS} */
  pour $j = 1 \rightarrow m$ **faire**
    si τ_i ordonnançable sur π_j **alors**
      /* τ_i est attribuée à π_j */ ;
      $\tau(\pi) = \tau(\pi) \cup \tau_i$;
      $Ordo \leftarrow vrai$;
      break ;
    fin si
  fin pour
  si not $Ordo$ **alors**
    /* Essai de fractionnement de la tâche */
    si $k < maxK$ **alors**
      /* La profondeur limite de fractionnement maxK n'est pas atteinte*/
      /* $\tau_i(O_i, C_i, T_i, D_i, s_i)$ est fractionnée en un ensemble composé de a sous-tâches */
      si $s_i = \infty$ **alors**
        $\tau_i^{split} = \{\tau_i^0(O_i + x \times T_i, C_i, 2 \times T_i, D_i, \infty), x = 0..1\}$; // $a = 2$
      sinon
        si $s_i = 2$ **alors**
          $\tau_i^{split} = \{\tau_i^0(O_i + x \times s_i T_i, C_i, 2 \times s_i T_i, D_i, \infty), x = 0..1\}$; // $a = 2$
        sinon
          $\tau_i^{split} = \{\tau_i^x(O_i + x \times T_i, C_i, s_i \times T_i, D_i, \infty), x = 0..(s_i - 2)\}$ // $a = s_i - 1$
        fin si
      fin si
      si $KTS - FF_{QoS}(m, k+1, maxK, \tau_i^{split}) ==$ SUCCES **alors**
        $Ordo \leftarrow vrai$;
      fin si
    sinon
      /* La profondeur limite de fractionnement maxK a été atteinte*/
      $Ordo \leftarrow faux$;
    fin si
  fin si
  si not $Ordo$ **alors**
    retourner ECHEC ;
  fin si
fin pour
retourner SUCCES ;
fin

FIGURE 3.4 – Pseudo-code de $KTS - FF_{QoS}$

Le fonctionnement de $KTS - AA_{QoS}$ détermine, par des applications répétées du Lemme 3.1 au niveau de chaque tâche à assigner, l'ordonnançabilité de l'ensemble de tâches.

Théorème 21 *Considérant un ensemble de tâches périodiques τ, si l'algorithme de partitionnement $KTS - AA_{QoS}$ retourne SUCCES, alors le partitionnement établi est ordonnançable.*

Preuve: On observe que l'algorithme retourne SUCCES si et seulement s'il a assigné avec succès chaque tâche de τ aux différents processeurs. Avant l'assignation de la tâche τ_i, l'ensemble

de tâches de chaque processeur est trivialement ordonnançable. Il résulte du Lemme 3.1 que tous les ensembles de tâches des processeurs restent ordonnançables après chaque attribution de tâche. Par conséquent, tous les ensembles de tâches des processeurs sont ordonnançables une fois que toutes les tâches de τ ont été assignées.□

3.3 Analyse de $KTS - AA_{QoS}$

Nous cherchons à présent à établir quelle est la profondeur limite de l'algorithme $KTS - AA_{QoS}$. Nous avons étudié dans le chapitre 2 la profondeur limite de $KTS - AA$ correspondant au moment à partir duquel le nombre de tâches dérivant d'une tâche τ_s devient supérieur ou égal au nombre de processeurs ; il en est de même pour $KTS - AA_{QoS}$.

Lemme 3.2 *Au pire-cas, la profondeur maximale $maxK$ pouvant être atteinte par l'algorithme $KTS - AA_{QoS}$ pour fractionner une tâche τ_s en m sous-tâches est telle que $maxK = \max(1, m + 1 - a)$ (avec $a = 2$ si $s_i = 2; \infty$, $a = s_i - 1$ autrement).*

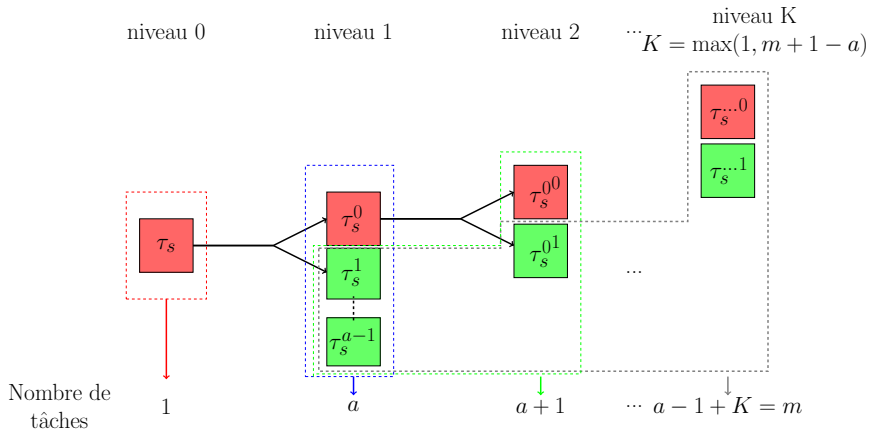


FIGURE 3.5 – Cas 1 : Profondeur maximale atteinte au pire-cas pour le fractionnement de τ_s en m sous-tâches

Preuve:

La Figure 3.5 illustre le pire scénario, pour avoir m sous-tâches dérivantes d'une tâche τ_s sur une profondeur maximale $maxK$. Si sur chaque niveau, une des tâches fractionnées est rejetée cela nous conduit à avoir au niveau K un nombre de tâches dérivantes égal à $a - 1 + (K - 2) + 2$, soit $a + K - 1$. Comme nous souhaitons avoir m tâches dérivantes au maximum (car la plateforme comporte m processeurs) alors la profondeur maximale $maxK$ doit être dimensionnée à $m + 1 - a$. Nous veillons toutefois à autoriser au moins un niveau de fractionnement lorsque $m + 1 - a \leq 0$ d'où $maxK = \max(1, m + 1 - a)$. □

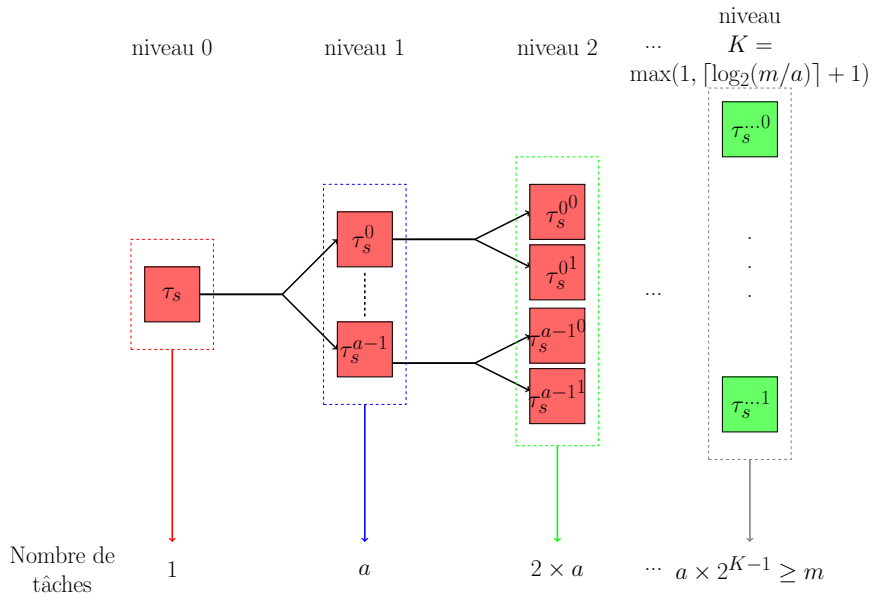


FIGURE 3.6 – Cas 2 : Profondeur minimale pour le fractionnement de τ_s en m sous-tâches

Lemme 3.3 *La profondeur minimale $minK$ nécessaire pour fractionner une tâche τ_s en m sous-tâches est telle que : $minK = \max(1, \lceil \log_2(m/a) \rceil + 1)$.*

Preuve: Pour avoir m sous-tâches dérivantes d'une tâche τ_s sur une profondeur minimale, il faut que sur chaque niveau toutes les nouvelles tâches soient fractionnées. Cela nous conduit à avoir au niveau K un nombre de tâches dérivantes égal à $a.2^{K-1}$. Comme nous souhaitons avoir m tâches dérivantes au maximum alors la limite de profondeur minimale $minK$ doit être égale à $\max(1, \lceil \log_2(m/a) \rceil + 1)$ (idem, si $a \geq m+1$, on autorise au moins un niveau de fractionnement). □

Théorème 22 *La complexité de $KTS - AA_{QoS}$ est en $O(n^2)$.*

Preuve: Considérons un ensemble de n tâches à ordonnancer sur une plateforme à m processeurs :

- Si $n \leq m$, il n'y a pas de task splitting comme dans le pire des cas chaque tâche passe sur un processeur.
  - avec $KTS - FF_{QoS}$: la première tâche passe le test sur le premier processeur et y est assignée (1 opération), la seconde passe son test au pire cas sur le premier et le second processeur (2 opérations), ..., la n-ième tâche passe le test au pire-cas sur les n processeurs avant d'être assignée (n opérations). Ce qui nous amène à un nombre d'opérations total égal à $1 + ... + n$ soit $n(n+1)/2$.
  - avec $KTS - BF_{QoS}/KTS - WF_{QoS}$: chaque tâche passe le test d'ordonnançabilité sur les m processeurs avant de choisir celui qui peut la recevoir tout en maximisant sa capacité inutilisée. Le nombre d'opérations total est donc égal à $n.m$.
- Si $n > m$ et considérant que s_i est le même pour toutes les tâches :
  - avec $KTS - FF_{QoS}$: les m premières tâches passent leur test comme décrit dans le cas $n \leq m$ ($1 + 2 + ... + m = (m+1)m/2$ opérations) et sont toutes attribuées avec succès. Le reste des tâches $(n-m)$ passe au pire cas son test à chaque fois sur les m processeurs ($(n-m)m$ opérations). Si ces dernières n'arrivent pas à être assignées, elles sont splittées en un nombre donné a qui est fonction de s_i ($(s_i = 2 || s_i = \infty) \Rightarrow a = 2$ nouvelles tâches, sinon $a = s_i - 1$) au premier niveau de task splitting ($K = 1$). Ensuite, à chaque niveau k, leur nombre doublent. L'algorithme essaie d'assigner ces nouvelles tâches sur les m processeurs ($a.(n-m)m + a.(n-m)m.2 + ... + a.(n-m)m.2^{K-1}$ opérations). Le nombre

total d'opérations est donc égal à $m(m+1)/2 + a.(n-m)m \times K_s$ avec $K_s = \sum_{k=0}^{K} 2^{k-1}$ ayant une valeur constante.

– avec $KTS-BF_{QoS}/KTS-WF_{QoS}$: les m premières tâches passent leur test chacune sur les m processeurs comme décrit dans le cas de $n \leq m$ (m^2 opérations) et sont toutes attribuées avec succès. Les tâches qui suivent $(n-m)$ passent leur test sur les m processeurs ($(n-m)m$ opérations). Si ces dernières n'ont pu à être assignées, elles sont splittées à chaque niveau k comme décrit précédemment pour $KTS-FF_{QoS}$. Le nombre total d'opérations est donc égal à $m^2 + a.(n-m)m \times K_s$ avec $K_s = \sum_{k=0}^{K} 2^{k-1}$ ayant une valeur constante.

Comme $n = \beta m$ avec $\beta \epsilon \mathbb{R}^+$, la complexité de $KTS-AA_{QoS}$ se trouve être en $O(n^2)$.

4 Validation par simulation

L'objectif de ce paragraphe est de décrire les résultats des simulations effectuées dans le but de tirer profit des heuristiques adaptées décrites précédemment et d'analyser leur comportement face aux variations de plusieurs paramètres (ex : charge du système, taux de pertes autorisées, ...). L'objectif est de maximiser le taux de réussite, c'est-à-dire la proportion d'ensembles de tâches générés qui s'exécutent dans le respect de leurs contraintes temporelles.

4.1 Environnement de simulation

L'environnement de simulation consiste en m ordonnanceurs monoprocesseur associés aux différents processeurs constituant la plateforme. Le programme de simulation a été implémenté en langage Perl. L'architecture fonctionnelle du simulateur est représentée sur la Figure 3.7.

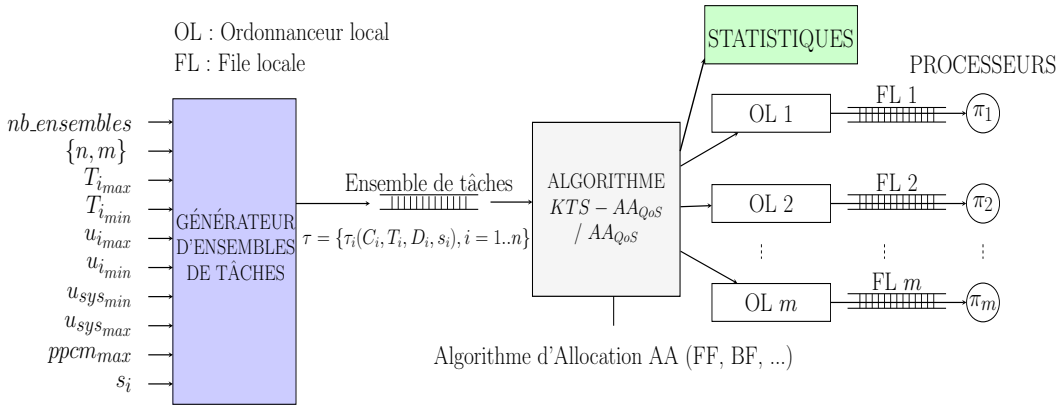


FIGURE 3.7 – Architecture fonctionnelle du simulateur

Le générateur d'ensembles de tâches périodiques a été conçu de manière à ce que les facteurs d'utilisation u_i des tâches de l'ensemble généré soient répartis de manière uniforme dans $[u_{i_{min}}, u_{i_{max}}] = [0.1, 1]$. Il accepte en entrée plusieurs paramètres tels que : le nombre de tâches n souhaité pour la configuration, le ppcm maximum $ppcm_{max}$ des périodes des tâches choisies dans l'intervalle $[T_{i_{min}}, T_{i_{max}}] = [10, 200]$ et les facteurs de pertes s_i des tâches. La charge périodique totale du système (composé de m processeurs) u_{sys} varie dans $[u_{sys_{min}}, u_{sys_{max}}]$.

En sortie, on obtient pour chaque valeur de u_{sys}, *nb_ensembles* ensembles de tâches périodiques fermes $\tau = \{\tau_i(0, C_i, T_i, D_i, s_i), i = 1..n\}$ (tâches synchrones donc $O_i = 0$). Les durées d'exécution des tâches sont générées à partir des facteurs d'utilisation et des périodes affectés aux tâches, soit $C_i = u_i \times T_i$. D_i choisi aléatoirement dans $[C_i, T_i]$ ($D_i \leq T_i$).
Après la génération des différents ensembles de tâches, le simulateur tente d'ordonnancer, pour

chaque valeur de u_{sys}, les *nb_ensembles* ensembles de tâches et détermine le taux de réussite qui permettra de comparer les différentes stratégies entre elles.

Dans toutes nos simulations, nous générons 50 ensembles de 20 tâches sur une plateforme composé de 8 processeurs avec un facteur d'utilisation du système u_{sys} variant dans l'intervalle $[0.6, 1.3]$. Nos observations se font à chaque fois sur (i) des ensembles composés de tâches à échéances contraintes ($D \leq T$) et (ii) des ensembles composés de tâches à échéances sur requêtes ($D = T$).

4.2 Résultats de simulation

4.2.1 Effet de la profondeur limite K pour $KTS - AA_{QoS}$

L'algorithme $KTS - AA_{QoS}$ est comparé à l'heuristique de partitionnement classique sous contraintes de QoS AA_{QoS}. $KTS - AA_{QoS}$ ($K = x$) correspond à l'algorithme KTS possédant une profondeur limite de task splitting $K = x$ et utilisé avec l'heuristique de partitionnement AA_{QoS}.

Cas 1 : $D = T$
Les Figures 3.8, 3.9 et 3.10 illustrent les performances de $KTS - AA_{QoS}$ avec $D = T$ pour différentes valeurs de K et lorsque BF_{QoS}, BF_{QoS} et BF_{QoS} sont respectivement considérées.

Comme prévu, lorsque la profondeur K augmente, les performances de $KTS - AA_{QoS}$ augmentent et dépassent celles de AA_{QoS} quelle que soit l'heuristique de base utilisée et le facteur de pertes s_i. Nous observons par exemple, lorsque le système est en surcharge (ex : $u_{sys} = 1.2$ et $s_i = 2$ voir Figure 3.9(a)), FF_{QoS} arrive à ordonnancer seulement 10% des ensembles de tâches générés tandis que 26% le sont en utilisant $KTS - FF_{QoS}$ avec $K = 1$, 26% en utilisant $KTS - FF_{QoS}$ avec $K = 2$ et jusqu'à 28% en utilisant $KTS - FF_{QoS}$ avec $K = 3$.
Par ailleurs, il est intéressant de noter dans cet exemple que le fait de tolérer certaines pertes permet de trouver un partitionnement valide pour des ensembles de tâches ayant un facteur d'utilisation dépassant la capacité du système.

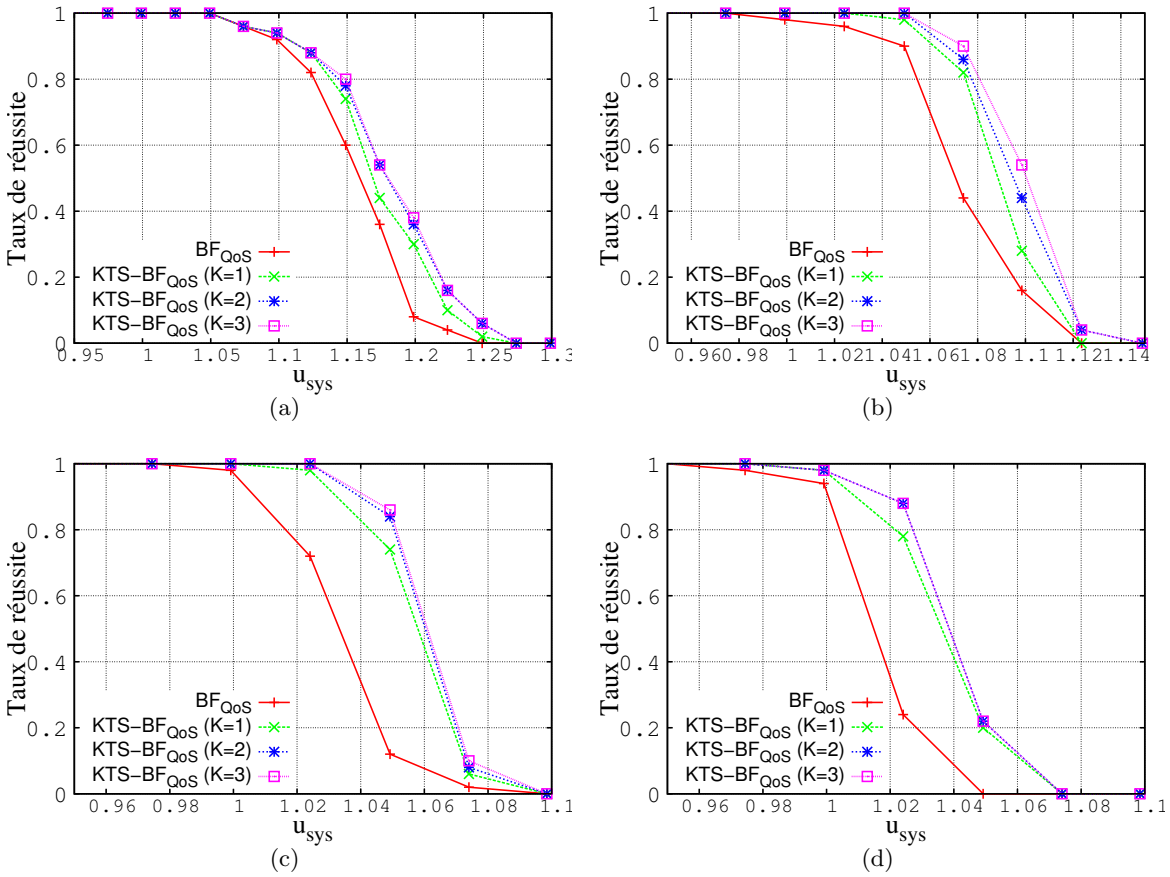


FIGURE 3.8 – Performances de BF_{QoS} et $KTS - BF_{QoS}$ avec $D = T$ pour (a) $s_i = 2$ (50% de pertes autorisées), (b) $s_i = 3$ (33% de pertes autorisées) , (c) $s_i = 4$ (25% de pertes autorisées) et (d) $s_i = 5$ (20% de pertes autorisées)

De plus, nous remarquons que pour chaque s_i, $KTS–AA_{QoS}$ converge à partir d'une certaine valeur. Cette limite étant dûe au fait que la tâche a été fractionnée sur m processeurs ou plus, le fractionnement devient inutile et les performances cessent d'augmenter.

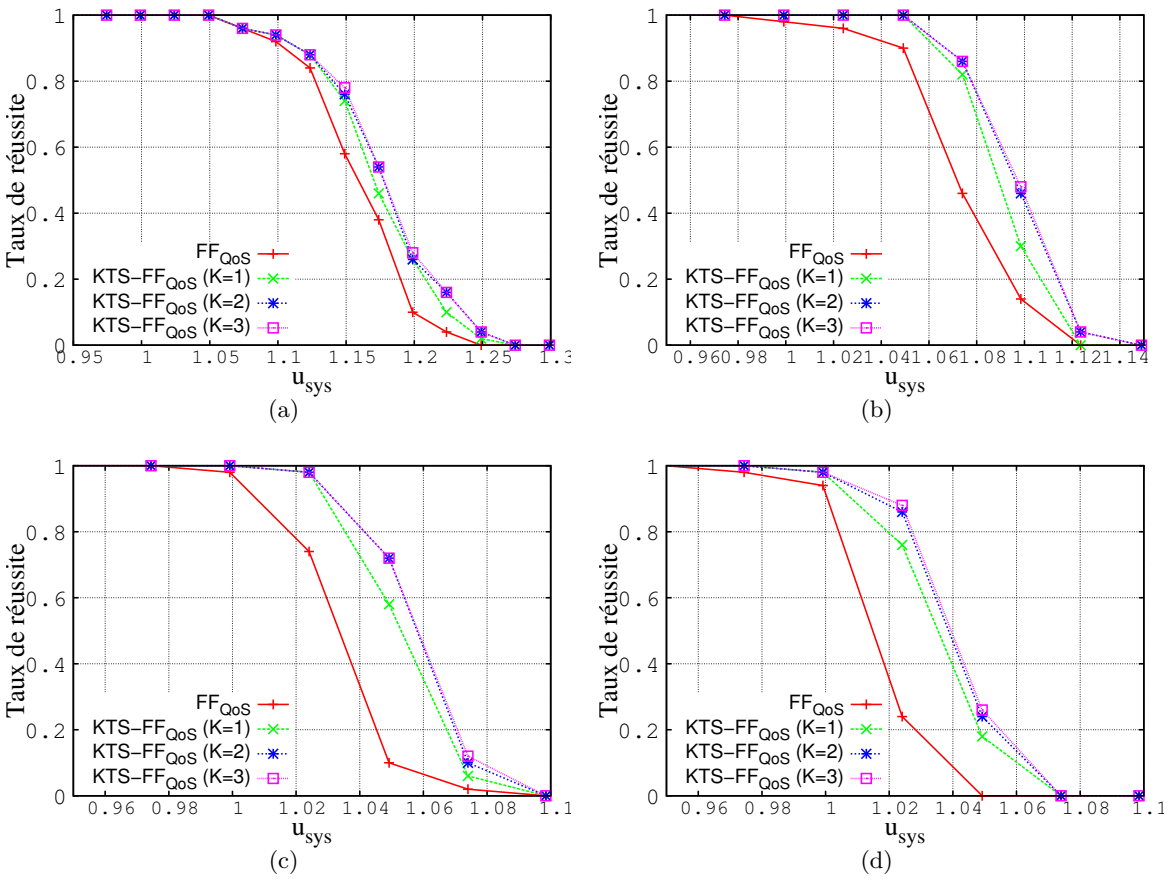


FIGURE 3.9 – Performances de FF_{QoS} et $KTS - FF_{QoS}$ avec $D = T$ pour (a) $s_i = 2$ (50% de pertes autorisées), (b) $s_i = 3$ (33% de pertes autorisées) , (c) $s_i = 4$ (25% de pertes autorisées) et (d) $s_i = 5$ (20% de pertes autorisées)

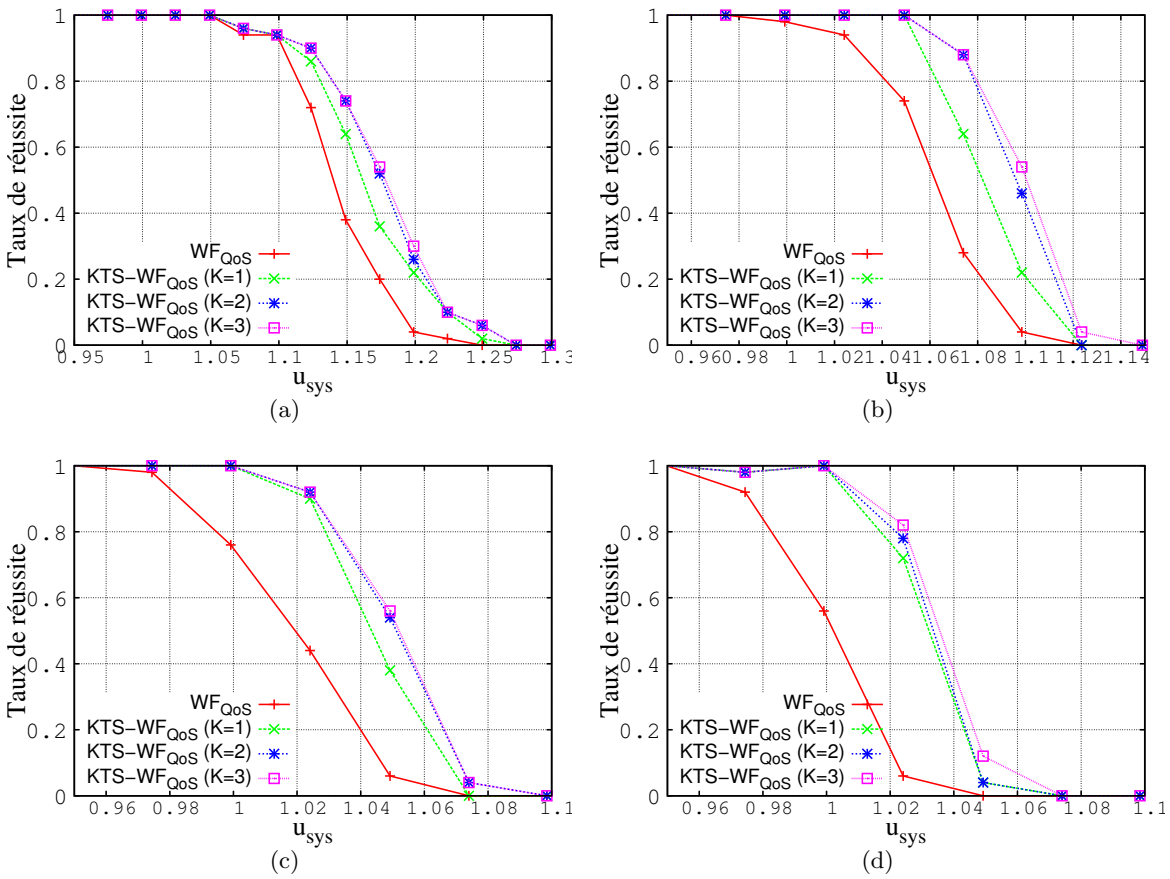


FIGURE 3.10 – Performances de WF_{QoS} et $KTS-WF_{QoS}$ avec $D = T$ pour (a) $s_i = 2$ (50% de pertes autorisées), (b) $s_i = 3$ (33% de pertes autorisées) , (c) $s_i = 4$ (25% de pertes autorisées) et (d) $s_i = 5$ (20% de pertes autorisées)

Cas 2 : $D \leq T$

Les Figures 3.11, 3.12 et 3.13 illustrent les performances de $KTS-AA_{QoS}$ pour différentes valeurs de K avec $D \leq T$ et lorsque BF_{QoS}, BF_{QoS} et BF_{QoS} sont respectivement considérées.

Pour le modèle de tâches à échéances contraintes, les conclusions sont les mêmes que pour le cas de tâches à échéances sur requêtes : lorsque la profondeur K augmente, les performances de $KTS-AA_{QoS}$ augmentent et dépassent celles de AA_{QoS} quelle que soit l'heuristique de base utilisée et le facteur de pertes s_i. Par exemple, lorsque le système est fortement chargé (ex : $u_{sys} = 1.05$ et $s_i = 2$ voir Figure 3.12(a)), FF_{QoS} arrive à ordonnancer seulement 10% des ensembles de tâches générés tandis que 12% le sont en utilisant $KTS-FF_{QoS}$ avec $K = 1$, et jusqu'à 14% en utilisant $KTS-FF_{QoS}$ avec $K = 3$.

De même, pour chaque s_i, $KTS-AA_{QoS}$ converge à partir d'une certaine valeur du fait que comme la tâche a été fractionnée en m processeurs ou plus, le fractionnement devient inutile et les performances cessent d'augmenter.

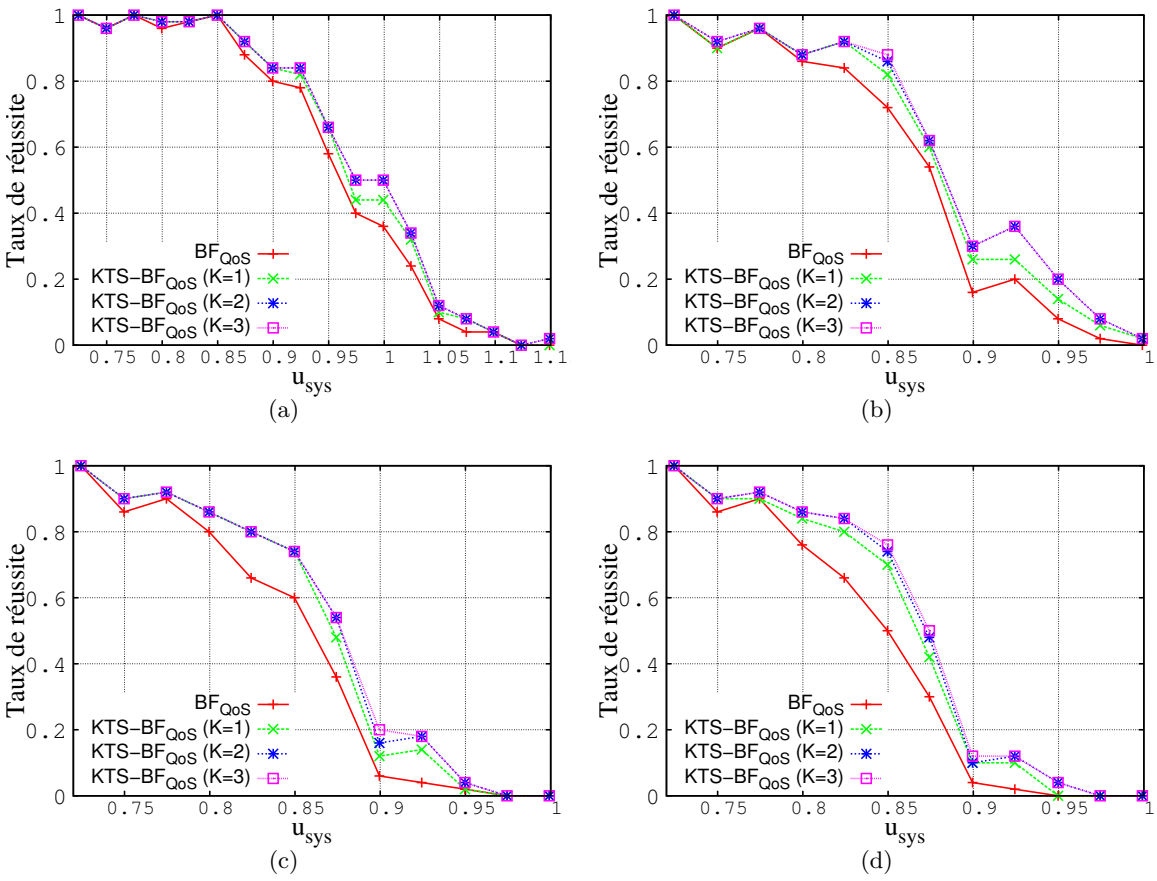


FIGURE 3.11 – Performances de BF_{QoS} et $KTS - BF_{QoS}$ avec $D \leq T$ pour (a) $s_i = 2$ (50% de pertes autorisées), (b) $s_i = 3$ (33% de pertes autorisées) , (c) $s_i = 4$ (25% de pertes autorisées) et (d) $s_i = 5$ (20% de pertes autorisées)

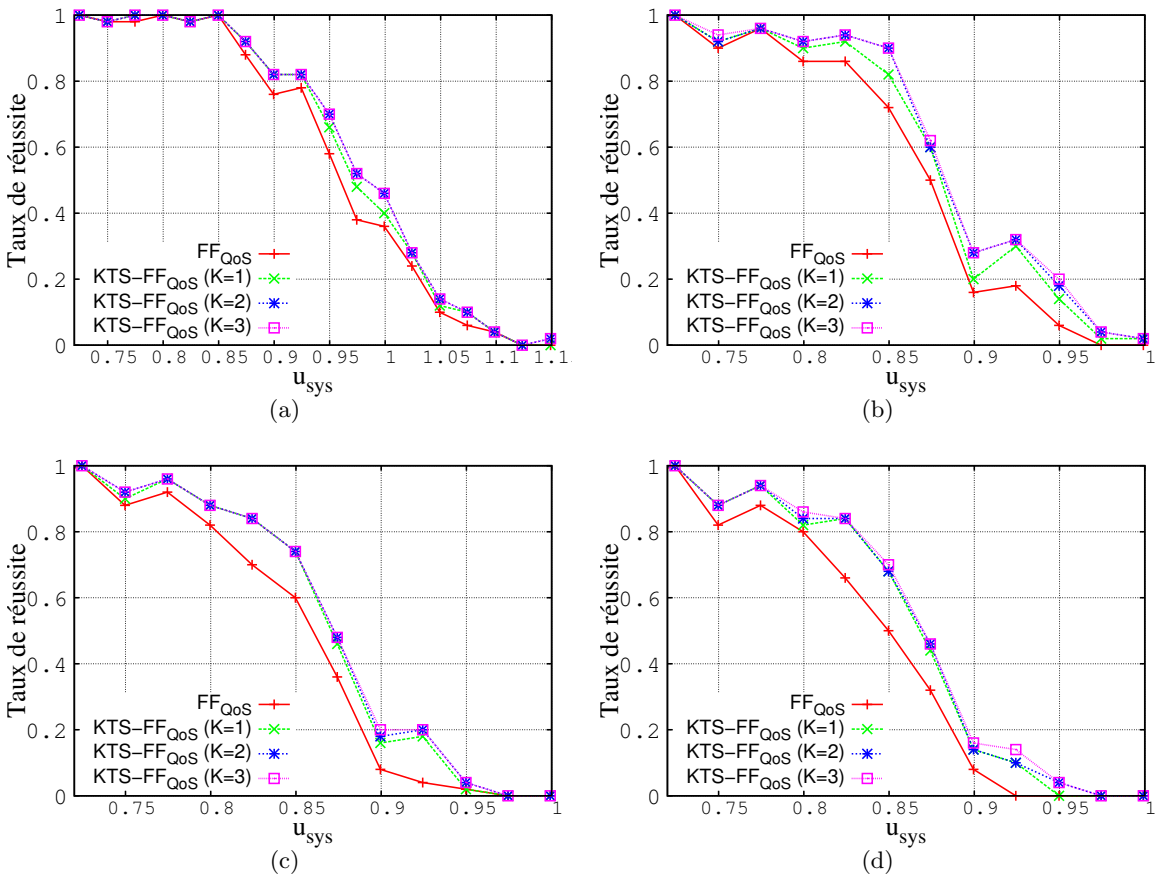


FIGURE 3.12 – Performances de FF_{QoS} et $KTS-FF_{QoS}$ avec $D \leq T$ pour (a) $s_i = 2$ (50% de pertes autorisées), (b) $s_i = 3$ (33% de pertes autorisées) , (c) $s_i = 4$ (25% de pertes autorisées) et (d) $s_i = 5$ (20% de pertes autorisées)

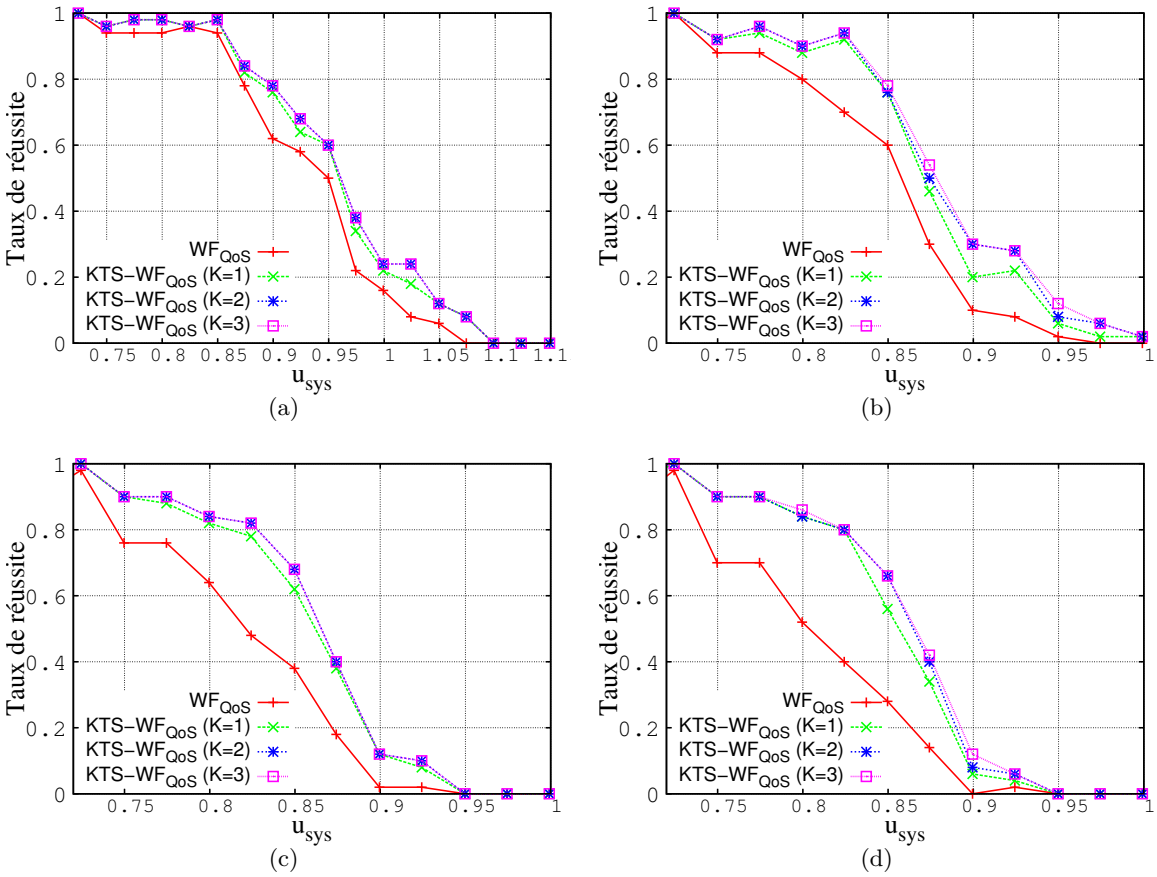


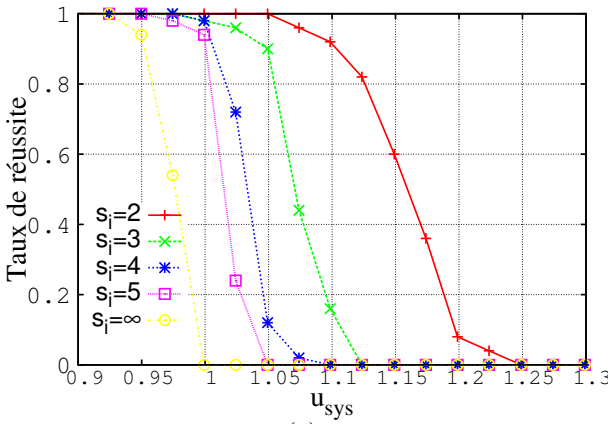
FIGURE 3.13 – Performances de WF_{QoS} et $KTS-WF_{QoS}$ avec $D \leq T$ pour (a) $s_i = 2$ (50% de pertes autorisées), (b) $s_i = 3$ (33% de pertes autorisées) , (c) $s_i = 4$ (25% de pertes autorisées) et (d) $s_i = 5$ (20% de pertes autorisées)

4.2.2 Effet du facteur de pertes s_i

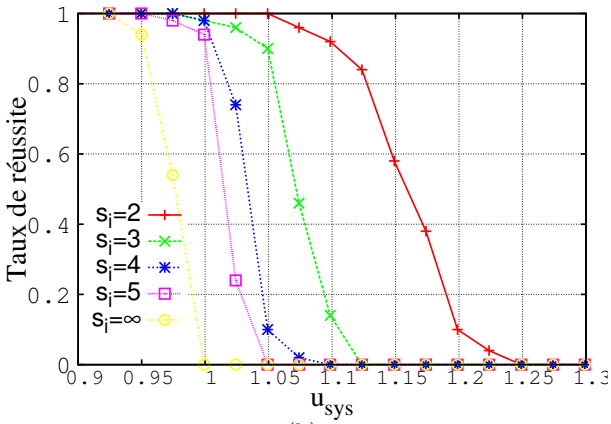
Cas 1 : $D = T$

La Figure 3.14 illustre l'effet du paramètre de pertes s_i sur les différents algorithmes BF_{QoS}, FF_{QoS} et WF_{QoS} pour un modèle de tâches à échéances sur requêtes.

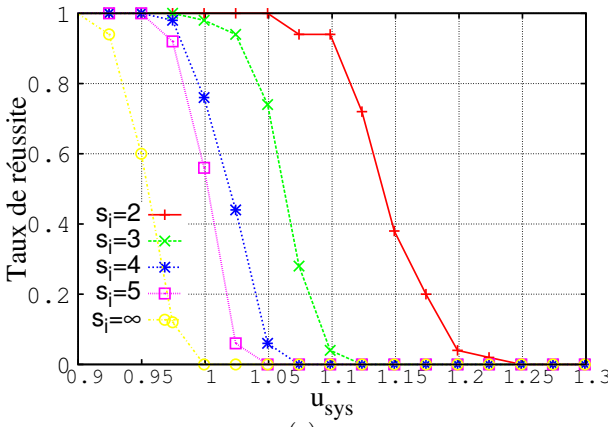
Nous pouvons noter dans la Figure 3.14 la baisse des performances de AA_{QoS} à mesure que s_i augmente. Cela est dû au fait que les pertes autorisées deviennent plus rares à mesure que s_i augmente. Par exemple, lorsque $u_{sys} = 1.05$, FF_{QoS} arrive à ordonnancer : (i) 100% des ensembles de tâches générés avec $s_i = 2$ tandis que seulement 18% le sont avec $s_i = 5$. La Figure 3.15 illustre le taux de réussite observé sous $KTS-AA_{QoS}$ avec $D \leq T$ en fonction des pertes autorisées témoignant aussi de la baisse des performances de AA_{QoS} à mesure que s_i augmente. En particulier, lorsque $u_{sys} = 1.05$, $KTS-FF_{QoS}$ arrive à ordonnancer : (i) 100% des ensembles de tâches générés avec $s_i = 2$ tandis que 26% le sont avec $s_i = 5$.



(a)



(b)



(c)

FIGURE 3.14 – Performances de AA_{QoS} avec $D = T$ en fonction de s_i (a) BF_{QoS}, (b) FF_{QoS} , (c) WF_{QoS}

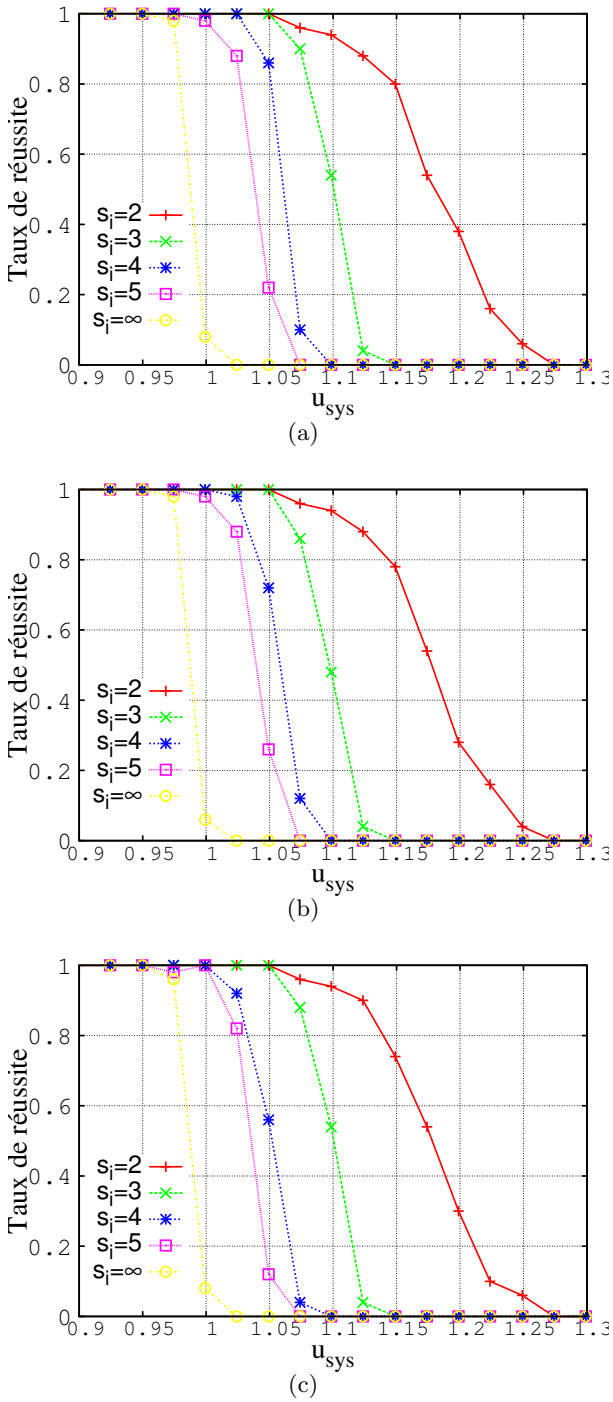
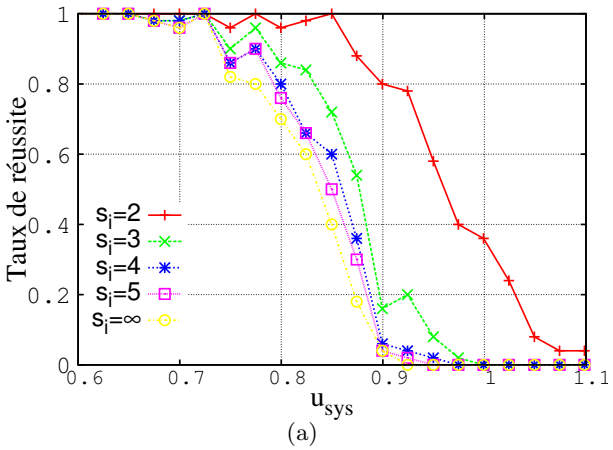


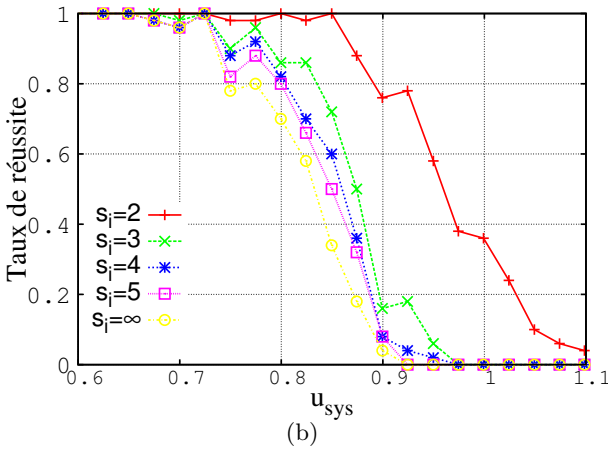
FIGURE 3.15 – Performances de $KTS - AA_{QoS}$ (K=3) avec $D = T$ en fonction de s_i (a) $KTS - BF_{QoS}$, (b) $KTS - FF_{QoS}$, (c) $KTS - WF_{QoS}$

Cas 2 : $D \leq T$

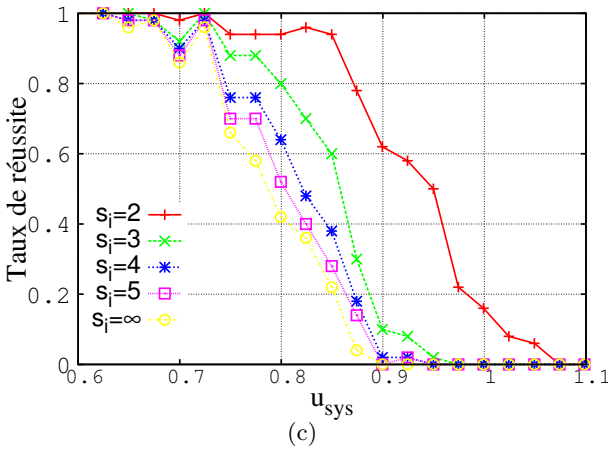
La Figure 3.16 illustre l'effet du paramètre de pertes s_i sur les différents algorithmes BF_{QoS}, FF_{QoS} et WF_{QoS} avec $D \leq T$.



(a)



(b)



(c)

FIGURE 3.16 – Performances de AA_{QoS} avec $D \leq T$ en fonction de s_i (a) BF_{QoS}, (b) FF_{QoS} , (c) WF_{QoS}

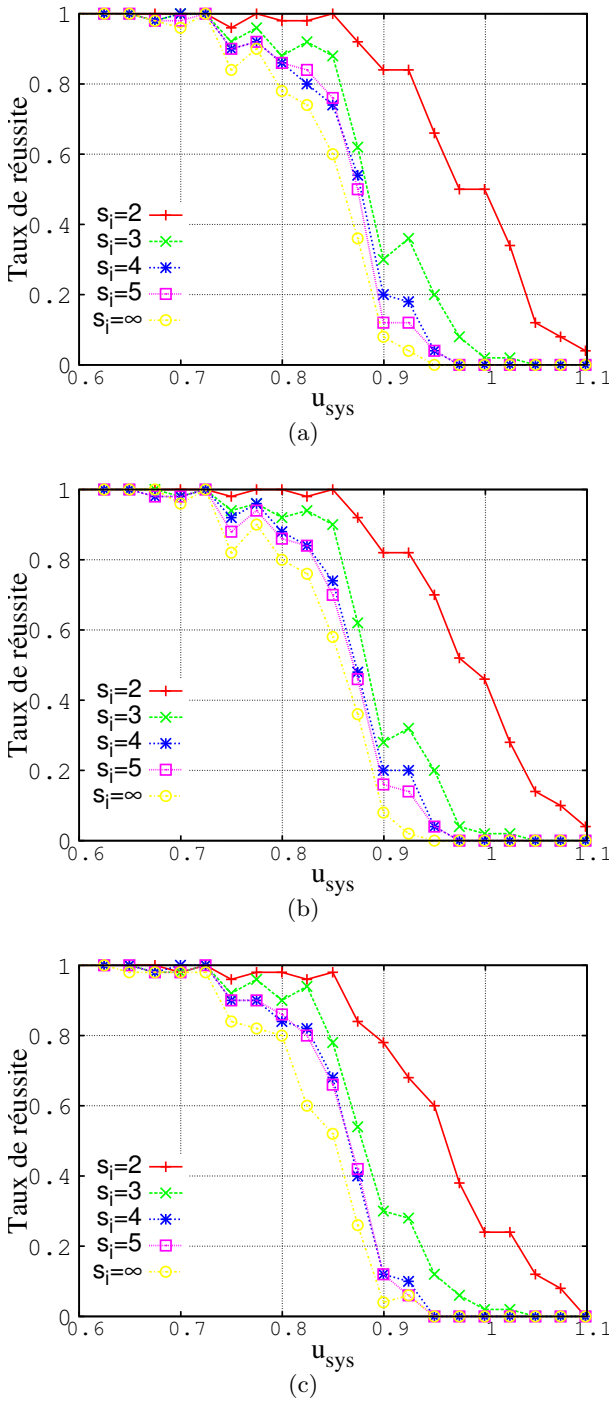


FIGURE 3.17 – Performances de $KTS-AA_{QoS}$ (K=4) avec $D \leq T$ en fonction de s_i (a) $KTS-BF_{QoS}$, (b) $KTS-FF_{QoS}$, (c) $KTS-WF_{QoS}$

Les conclusions sont quasiment identiques au cas $D = T$: dans la Figure 3.16, les performances de AA_{QoS} diminuent à mesure que s_i augmente. Cela étant lié au fait que les pertes autorisées deviennent plus rares à mesure que s_i augmente. Par exemple, lorsque $u_{sys} = 1$ avec $D \leq T$, FF_{QoS} arrive à ordonnancer : (i) 36% des ensembles de tâches générés avec $s_i = 2$ tandis que 0% le sont avec $s_i = 5$.

La Figure 3.17 illustre le taux de réussite observé sous $KTS-AA_{QoS}$ en fonction des pertes autorisées et de l'heuristique considérée avec $D \leq T$. Cette fois-ci, les performances ne sont pas systématiquement meilleures à mesure que l'on autorise plus de pertes. En particulier, lorsque $u_{sys} = 0.8$, $KTS-BF_{QoS}$ arrive à ordonnancer : 84% des ensembles de tâches générés avec $s_i = 2$

tandis que 98% le sont en utilisant avec $s_i = 3$, 80% avec $s_i = 4$, 84% avec $s_i = 5$, et 74% avec $s_i = \infty$. Il est clair que la valeur de s_i va déterminer le nombre de tâches dérivantes et par la suite le facteur par lequel u_i va être divisé. Un nombre plus important de tâches implique plus de nouvelles tâches avec un facteur d'utilisation plus petit et donc plus facilement assignables. Cependant dans le cas de $s_i = 5$, au deuxième niveau il devient inutile de fractionner comme le nombre de tâches devient égale à $m = 8$ alors que le taux de pertes autorisées continue à diminuer d'où les performances qui chutent par rapport à $s_i = 5$.
De manière générale, nous veillerons à fixer s_i en fonction de la plateforme et de l'utilisation du système. Par exemple, pour $D = T$, lorsque u_{sys} est :

- supérieur à 115%, nous veillerons à fixer s_i à 2.
- dans [110%,115%[, le concepteur, en fonction du taux de pertes qu'il peut autoriser, décidera entre $s_i = 2$ ou $s_i = 3$.
- dans [105%,110%[, le concepteur, en fonction du taux de pertes qu'il peut autoriser, décidera entre s_i =2, 3, 4 ou 5.

5 Conclusion

Ce chapitre a permis d'étendre les heuristiques de partitionnement classiques pour un modèle de tâches fermes. Nous avons abordé le problème de l'attribution d'un ensemble de tâches fermes dans un système multiprocesseur. Ensuite une analyse d'ordonnançabilité selon l'algorithme RTO a été présentée pour garantir le respect des échéances des jobs rouges des tâches synchrones de l'ensemble. Celle-ci a été étendue pour un modèle de tâches périodiques fermes asynchrones. Nous avons par la suite adapté l'approche de partitionnement temps réel basée sur le fractionnement à un modèle de tâches fermes (heuristique $KTS - AA_{QoS}$). Au travers de ce travail, nous avons montré comment l'analyse étendue d'ordonnançabilité développée dans le contexte de tâches périodiques asynchrones fermes peut être utilisée conjointement à un algorithme de fractionnement de tâches. L'évaluation sur une large gamme d'ensembles de tâches, générés de manière aléatoire, indique que l'algorithme $KTS - AA_{QoS}$ offre des performances significativement meilleures que celles des heuristiques de partitionnement AA_{QoS} et ce, quel que soit le taux de pertes autorisées. À travers un exemple de système, nous avons montré comment guider le concepteur à faire son choix au niveau des pertes autorisées en fonction de l'utilisation du système de manière à obtenir de bonnes performances.

Chapitre 4

Contribution au partitionnement sous contraintes énergétiques

Ce chapitre est consacré au problème relatif au partitionnement d'un ensemble de tâches temps réel périodiques sur une plateforme multiprocesseur homogène autonome du point de vue énergétique. Pour cela, nous décrivons dans un premier temps le système considéré, un système qui se doit de satisfaire ses contraintes temporelles sans pour autant manquer d'énergie pour exécuter ses tâches. Après avoir introduit le modèle considéré et formalisé le problème, nous présentons les conditions d'ordonnançabilité associées aux systèmes temps réel autonomes énergétiquement. Nous présentons ensuite une adaption des heuristiques de partitionnement classiques aux systèmes multiprocesseur homogènes autonomes énergétiquement. Notre objectif est de mettre en œuvre une méthode de partitionnement qui garantisse l'absence de famine énergétique tout en utilisant Earliest Deadline First (EDF) comme politique d'ordonnancement. Nous explorons la façon dont à la fois les critères de tri des tâches temps réel et les contraintes énergétiques peuvent favoriser la performance des différentes heuristiques. Dans un deuxième temps, nous proposons une extension de l'algorithme KTS (décrit dans le chapitre 2) à des systèmes multiprocesseur homogènes autonomes énergétiquement.

1 Modèles et définitions

1.1 Système considéré

Nous nous intéressons dans cette partie à la proposition de solutions de partitionnement pour des systèmes nomades qui par nature doivent être autonomes du point de vue énergétique. Comme ces systèmes sont le plus souvent embarqués en environnement hostile (voire inaccessible), limitant ainsi les interventions humaines, ils doivent fonctionner grâce à des réservoirs d'énergie, qui se rechargent continuement au cours du temps à partir d'une source d'énergie renouvelable. Cette problématique est principalement traitée par des méthodes de type DVS et/ou DPM visant à faire baisser la consommation d'énergie des circuits électroniques. Il est clair que ces techniques sont utiles aux systèmes autonomes en leur permettant d'utiliser des batteries de plus faible capacité, des panneaux photovoltaïques de plus petite taille, et permettent d'étendre les durées séparant deux recharges successives d'une batterie. Pourtant, ces recharges demeurent impératives. Ces techniques ne permettent donc pas à elles seules d'assurer un fonctionnement à l'infini et qualifié de neutre énergétiquement. La neutralité énergétique se définit ici par la propriété que posséde le système embarqué à fonctionner dans le respect de toutes ses contraintes temporelles en n'utilisant que l'énergie disponible dans le réservoir et sans jamais en manquer.

1.2 Modèle du système

Nous considérons dans cette partie un système embarqué entièrement autonome capable de fonctionner dans le respect de toutes ses contraintes temporelles et énergétiques. La problématique liée à la conception des systèmes autonomes porte sur chacune de ses trois composantes :

- Le récupérateur d'énergie (*energy harvester*, en anglais) dont le choix dépend de la nature de l'énergie environnementale, de la quantité d'énergie requise, etc.
- Le réservoir d'énergie tel que la batterie et/ou le supercondensateur dont le choix dépend des dynamiques du système, des contraintes de dimensionnement, du coût, etc.
- Le consommateur d'énergie représenté ici par le support d'exécution des tâches temps réel.

Le modèle du système est illustré sur la Figure 4.1. Nous considérons une plateforme multiprocesseur composée de m processeurs identiques (π_j dénote le j-ième processeur de la plateforme) où chaque processeur π_j consomme de l'énergie de façon indépendante depuis son propre réservoir d'énergie de capacité B_j. Une application typique peut être un réseau de capteurs sans fil constitué de plusieurs sous-systèmes possédant différentes consommations d'énergie, chaque capteur étant activé en fonction des différents modes de fonctionnement du réseau (c.-à-d. échantillonnage, transmission ou réception de données). À un instant t, chaque récupérateur d'énergie (par exemple un panneau photovoltaïque) récupère l'énergie à partir d'une source d'énergie (par exemple le soleil ou une lampe) et la convertit en énergie électrique stockée dans son réservoir d'énergie de capacité B_j. Il paraît donc évident qu'à aucun instant, l'énergie stockée dans le réservoir de π_j, dénotée $E_j(t)$, ne doit dépasser sa capacité B_j :

$$\forall t, E_j(t) \leq B_j \tag{4.1}$$

Si l'énergie stockée atteint la capacité du réservoir, l'énergie récupérée entrante ne peut être stockée dans le réservoir et est donc rejetée. De même, le réservoir est considéré complètement vide lorsqu'il atteint un niveau minimum $E_{min} = 0$.

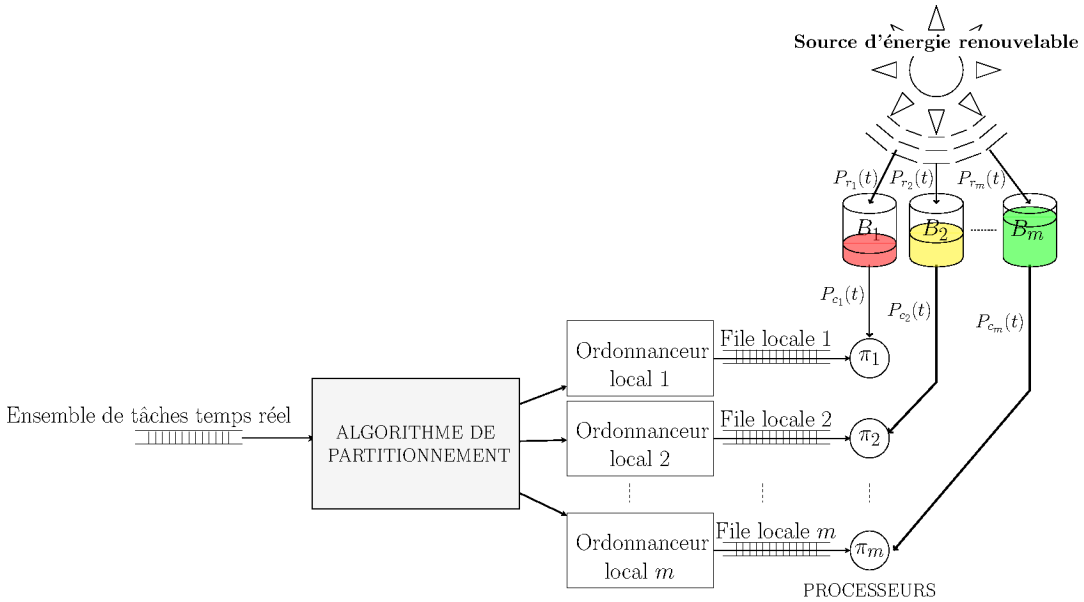


FIGURE 4.1 – Illustration du système autonome considéré

Nous noterons $P_{r_j}(t)$, la puissance instantanée de récupération d'énergie propre au réservoir de π_j. Nous supposerons que cette grandeur inclut toutes les pertes d'énergie occasionnées par la mise en œuvre matérielle. Nous appellerons *profil énergétique de la source* par rapport à chaque processeur π_j, la description de la fonction $P_{r_j}(t)$ pour toute la durée de vie de l'application, soit dans l'idéal un temps infini.

De plus, chaque processeur π_j du système embarqué est caractérisé par une puissance de consommation instantanée notée $P_{c_j}(t)$, exprimée en watts soit $0 \leq P_{c_j}(t)$. Cette puissance $P_{c_j}(t)$ est supposée nulle lorsque le processeur π_j est inactif.

L'énergie récupérée dans un intervalle $[t_1, t_2]$ dans le réservoir qui alimente le processeur π_j, notée $E_{r_j}(t_1, t_2)$ se calcule donc de la façon suivante :

$$E_{r_j}(t_1, t_2) = \int_{t_1}^{t_2} P_{r_j}(t).dt \tag{4.2}$$

En considérant la puissance instantanée de récupération d'énergie propre au réservoir de π_j constante ($\forall t, P_{r_j}(t) = P_{r_j}$), nous obtenons :

$$E_{r_j}(t_1, t_2) = (t_2 - t_1).P_{r_j} \tag{4.3}$$

À un instant t, l'énergie stockée dans le réservoir propre au processeur π_j est donnée par :

$$E_j(t) = \min(B_j, E_j(t-1) + E_{r_j}(t-1, t) - E_{c_j}(t-1, t)) \tag{4.4}$$

où $E_{c_j}(t-1, t)$ correspond à l'énergie requise entre $t-1$ et t pour l'exécution des jobs selon EDF relatifs aux tâches assignées au processeur π_j.

De plus, l'énergie stockée dans chaque réservoir est supposée être maximale à l'instant de démarrage du système (supposé à $t = 0$).

$$\forall j \epsilon 1..m, E_j(0) = B_j \tag{4.5}$$

où m représente le nombre de processeurs dans la plateforme.

1.3 Modèle de tâches

Nous considérons une plateforme π composée de m processeurs identiques : $\Pi = \{\pi_1, ..., \pi_m\}$, et un ensemble de n tâches indépendantes, périodiques et préemptibles $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Chaque tâche $\tau_i\,(O_i, E_i, C_i, T_i, D_i)$ est définie par :

- O_i, l'offset de τ_i ;
- E_i, sa consommation énergétique pire-cas (WCEC : Worst Case Energy Consumption), exprimée en joules ;
- C_i, son temps d'exécution pire-cas ;
- T_i, sa période ;
- D_i, son délai critique ou échéance relative.

Nous nous intéressons au modèle de tâches à échéances contraintes ($D_i \leq T_i$).

La consommation énergétique de τ_i est considérée au pire cas et correspond donc à la plus grande quantité d'énergie qu'elle peut consommer en s'exécutant sur un processeur. Le facteur d'utilisation ou charge processeur de la tâche τ_i, dénoté u_i, correspond au taux d'activité du processeur dédié à l'exécution des jobs successifs de la tâche : $u_i = \frac{C_i}{T_i}$. Le facteur d'utilisation d'un système composé de m processeurs et n tâches périodiques est défini comme suit :

$$u_{sys} = \frac{1}{m} \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{4.6}$$

Par ailleurs, nous introduisons le facteur d'utilisation énergétique $u_{e_i} = \frac{E_i}{T_i}$ de chaque tâche τ_i et sa densité énergétique $\delta_{e_i} = \frac{E_i}{\min(D_i, T_i)}$. Le facteur d'utilisation énergétique d'un ensemble de tâches τ est alors défini comme suit : $U_e = \sum_{i=1}^{n} u_{e_i}$.

Chaque processeur π_j est destiné à l'exécution du sous-ensemble de tâches qui lui sont assignées, soit Γ_j. Nous noterons $H(\Gamma_j) = PPCM(T_i)$ l'hyperpériode des tâches de l'ensemble Γ_j. Les surcoûts en temps et en énergie, occasionnés par les éventuelles préemptions, sont supposés nuls.

1.4 Éléments de terminologie énergétique

Nous proposons, pour ce nouveau modèle de système, d'introduire une nouvelle terminologie adaptée aux caracteristiques énergétiques des tâches, du processeur et de la source d'alimentation.

Définition 4.1 *La criticité d'énergie dénotée par R_e, appelée également ratio d'énergie, représente le niveau de contrainte énergétique du système. Il s'agit du rapport entre l'énergie consommée par les tâches de τ et l'énergie reçue par le système composé de m processeurs, soit :*

$$R_e = \frac{U_e}{\sum_{j=1}^{m} \overline{P}_{r_j}} \tag{4.7}$$

où $\overline{P}_{r_j} = \frac{1}{t} \int_0^t P_{r_j}(t).dt$.

Dans le cadre de cette thèse, nous considérerons que les *profils énergétiques de la source* pour chaque processeur π_j sont identiques, $\forall j = 1..m, P_{r_j}(t) = P_r(t)$. La formule de R_e se réduit à :

$$R_e = \frac{U_e}{m\overline{P}_r} \tag{4.8}$$

Nous classifierons par la suite le système selon trois catégories de criticité d'énergie : (i) faiblement contraint énergétiquement (*weakly energy-constrained* en anglais), (ii) moyennement contraint énergétiquement (*moderately energy-constrained* en anglais), et (iii) fortement contraint énergétiquement (*highly energy-constrained* en anglais).

Définition 4.2 *Un système est considéré faiblement contraint énergétiquement s'il consomme moins de 40% de l'énergie reçue par le système :* $R_e < 0.4$

Définition 4.3 *Un système est considéré moyennement contraint énergétiquement s'il consomme entre de 40% et 80% de l'énergie reçue par le système :* $0.4 \leq R_e \leq 0.8$

Définition 4.4 *Un système est considéré fortement contraint énergétiquement s'il consomme plus que 80% de l'énergie reçue par le système :* $R_e > 0.8$

- **Ordonnancement énergétiquement ordonnançable** : Nous disons qu'un ordonnancement est énergétiquement ordonnançable selon EDF pour un ensemble de tâches donnée si EDF produit une séquence d'ordonnancement où à aucun moment il ne se produit de famine énergétique.
- **Ordonnancement temporellement ordonnançable** : Nous disons qu'un ordonnancement est temporellement ordonnançable selon EDF pour un ensemble de tâches donnée si EDF produit une séquence valide d'ordonnancement où toutes les contraintes temporelles de cette configuration sont respectées sans prendre en compte ses contraintes énergétiques.
- **Ensemble de tâches ordonnançable / non-ordonnançable** : Nous disons qu'un ensemble de de tâches est ordonnançable selon une certaine politique d'ordonnancement si la séquence produite par l'ordonnanceur valide les contraintes temporelles et énergétiques de cet ensemble étant données les caractéristiques du réservoir d'énergie et de la source d'énergie. Sinon, elle est dite non-ordonnançable soit à cause de ses contraintes temporelles (échéances trop strictes) soit à cause de ses contraintes énergétiques (réservoir trop petit ou puissance de la source trop faible).

2 Le partitionnement sous contraintes temps réel et énergétiques

2.1 Formulation du problème

Le problème soulevé ici consiste à trouver un partitionnement *valide* c'est-à-dire satisfaisant les contraintes temporelles et énergétiques. Celui-ci permettra de déterminer la répartition d'un ensemble des tâches $\tau = \{\tau_1, \tau_2, ...\tau_n\}$ sur les différents processeurs de la plateforme $\Pi = \{\pi_1, ..., \pi_m\}$, de manière à ce que la configuration globale soit ordonnançable. Plus formellement, il s'agit de partitionner l'ensemble des n tâches en m sous-ensembles disjoints $\Gamma_1, \Gamma_2, ..., \Gamma_m$ (avec $\bigcup \Gamma_j = \tau$) et d'ordonnancer ensuite chaque sous-ensemble Γ_j sur le processeur π_j selon la politique d'ordonnancement monoprocesseur EDF en respectant :

- les contraintes d'ordre temporel : Sur chaque processeur π_j, tous les jobs de tâches de Γ_j doivent s'exécuter avant leur échéance.
- les contraintes d'ordre énergétique : Chaque processeur π_j ne doit, à aucun instant, manquer d'énergie pour exécuter l'ensemble des jobs de tâches de Γ_j.

Considérons un ensemble $\tau = \{\tau_1(0, 90, 30, 100, 100), \tau_2(0, 40, 5, 25, 25), \tau_3(0, 12, 36, 50, 50)\}$ sur une plateforme à 2 processeurs et une puissance de récupération d'énergie moyenne $\overline{P}_r = 2$. Le facteur d'utilisation de l'ensemble de tâches est égal à 1.22 tandis que le niveau de criticité $R_e = U_e/(2 * \overline{P}_r) = 0.725$. Une approche de partitionnement FF classique abstraction faite de la contrainte énergétique donnerait la répartition suivante (cf. Tableau 4.1) :

τ_i	π_j
τ_1	π_1
τ_2	π_1
τ_3	π_2

TABLE 4.1 – Exemple du résultat d'allocation de l'ensemble de tâches τ selon FF

L'approche de partitionnement FF-AS (décrite plus loin page 87) prenant en compte à la fois les contraintes temporelles et énergétiques donne quant à elle la répartition suivante (cf. Tableau 4.2) :

τ_i	π_j
τ_1	π_1
τ_2	π_2
τ_3	π_2

TABLE 4.2 – Exemple du résultat d'allocation de l'ensemble de tâches τ selon FF-AS

Comme le montre les Tableaux 4.1 et 4.2, les répartitions des tâches sont différentes pour un même ensemble en tenant compte ou non des contraintes énergétiques. Cela prouve donc qu'une répartition selon une heuristique faisant abstraction des contraintes énergétiques ne garantit pas systématiquement l'ordonnançabilité de l'ensemble d'où la nécessité de développer des extensions d'heuristiques de partitionnement qui répartissent les tâches selon leurs consommations énergétiques et le niveau de contrainte énergétique de la plateforme.

2.2 Tests d'ordonnançabilité

En faisant abstraction de la contrainte énergétique, une condition exacte d'ordonnançabilité pour EDF existe (cf Théorème 8 page 18) pour des ensembles de tâches périodiques à échéances sur requêtes (qu'elles soient synchrones ou asynchrones).

Pour des tâches périodiques synchrones à échéances contraintes, nous utiliserons les résultats énoncés dans [BRH90] qui reposent sur la notion de **demande processeur** (*demand bound function* en anglais). Dénotée $h_j(t)$ pour le processeur π_j, celle-ci correspond à la durée cumulée d'exécution des tâches réveillées et à terminer dans $[0,t]$:

Théorème 23 *Un ensemble Γ_j de n tâches périodiques synchrones à échéances contraintes est ordonnançable sur un processeur π_j par EDF si et seulement si :*

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

Pour un ensemble Γ_j de n tâches périodiques asynchrones, Leung and Merill [LM80] ont montré que l'intervalle de temps $[0, \max_{i=1..n}\{O_i\} + 2H(\Gamma_j)[$ est une fenêtre (pas nécessairement la plus courte) dans laquelle la charge de travail individuelle de chaque tâche est la même que celle calculée dans la fenêtre suivante (l'intervalle $[\max_{i=1..n}\{O_i\} + 2H(\Gamma_j), 2 \times (\max_{i=1..n}\{O_i\} + 2H(\Gamma_j))[$) et ainsi de suite, définissant ainsi une cyclicité (même charge de traitement). Ils ont donc montré qu'il suffisait de vérifier que toutes les échéances soient bien respectées dans l'intervalle $[0, \max_{i=1..n}\{O_i\} + 2H(\Gamma_j)]$ pour valider l'ordonnançabilité d'un ensemble de tâches. Baruah *et al.* ont utilisé ce résultat pour établir le théorème suivant :

Théorème 24 *[BRH90] Un ensemble Γ_j de n tâches périodiques asynchrones à échéances contraintes est faisable sur un processeur π_j si et seulement si :*

1. $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$.
2. $\forall 0 \leq t_1 < t_2 \leq \Phi + 2H(\Gamma_j), h_j(t_1, t_2) \leq t_2 - t_1$.

où $\Phi = \max_{i=1..n}\{O_i\}$ et $h_j(t_1, t_2)$ dénote la demande processeur relative à π_j dans l'intervalle $[t_1, t_2]$, à savoir :

$$h_j(t_1, t_2) = \sum_{i=1}^{n} \eta_i(t_1, t_2)\, C_i. \tag{4.10}$$

avec $\eta_i(t_1, t_2) = \max\left(0, \lfloor\frac{t_2 - O_i - D_i}{T_i}\rfloor - \lceil\frac{t_1 - O_i}{T_i}\rceil + 1\right)$

En rajoutant **la dimension énergétique** au problème, ces conditions sont étendues pour garantir les contraintes temporelles et d'énergie [Che13] :

Théorème 25 *[Che13] Un ensemble Γ_j de n tâches périodiques synchrones à échéances contraintes possédant un facteur d'utilisation énergétique U_{e_j} est ordonnançable sur un processeur π_j considérant un profil énergétique constant si et seulement si :*

1. $\forall 0 \leq t \leq H(\Gamma_j), h_j(t) \leq t$.
2. $\forall 0 \leq t \leq H(\Gamma_j), E_j(t) \geq 0$.
3. $E_j(H(\Gamma_j)) \geq E_j(0)$.
4. $U_{e_j} \leq P_{r_j}$

En effet, il a été prouvé récemment dans [Che13] qu'en considérant une source d'énergie à puissance constante, le Théorème 25 est une condition exacte et il est donc suffisant de la vérifier pour tous les instants t entre 0 et $H(\Gamma_j)$.

Dans le cas d'un ensemble de tâches asynchrones, il a été montré que la charge de traitement est la même d'une fenêtre à l'autre (de taille $\Delta = 2H(\Gamma_j) + \Phi$) [LM80]. Cependant, en partant du fait que les batteries sont initialement pleines et avec la présence d'offset, l'énergie reçue alors que le processeur est peu ou pas chargé au départ est perdu. Cependant, celle-ci est récupérée dans la deuxième fenêtre où la charge de traitement reste la même mais l'énergie reçue au début de la seconde fenêtre n'est plus perdue car utilisée (au moins une tâche active à $t = \Delta$). Pour cela, nous avançons une condition suffisante garantissant l'ordonnançabilité dans le cas d'un profil énergétique constant avec des batteries initialement pleines :

Théorème 26 *Un ensemble Γ_j de n tâches périodiques asynchrones à échéances contraintes possèdant un facteur d'utilisation énergétique U_{e_j} est ordonnançable sur un processeur π_j considérant un profil énergétique constant si :*

1. $\forall 0 \leq t_1 < t_2 \leq \Phi + 2H(\Gamma_j), h_j(t_1, t_2) \leq t_2 - t_1$.
2. $\forall 0 \leq t < 2 \times (2H(\Gamma_j) + \Phi), E_j(t) \geq 0$.
3. $E_j(2 \times (2H(\Gamma_j) + \Phi)) \geq E_j(0)$.

Preuve: (condition suffisante)
L'ensemble de tâches est ordonnançable sur $[2(k-1)\Delta, 2k\Delta]$, donc tous les jobs de tâches réveillés dans cet intervalle ont terminé leur exécution avant échéance (condition 1. [LM80]) sans famine énergétique ($E_j(t) \geq 0$). A l'instant $2k\Delta$, le système, du point de vue de l'activité du processeur se retrouve dans le même état qu'à $2(k-1)\Delta$. Du point de vue énergétique, le système se retrouve à l'instant $2k\Delta$ dans une situation plus favorable qu'à $2(k-1)\Delta$ avec un surplus d'énergie dans le réservoir à l'instant $2k\Delta$, égal à $E_j(2k\Delta) - E_j(2(k+1)\Delta)$. Par conséquent, l'ensemble sera aussi ordonnançable sur $[2k\Delta, 2(k+1)\Delta]$. Montrons maintenant que nous pouvons appliquer aux jobs réveillés sur $[2k\Delta, 2(k+1)\Delta]$ le même ordonnancement que celui appliqué aux jobs réveillés sur $[2(k-1)\Delta, 2k\Delta]$ et nous aurons alors nécessairement $E_j(2(k+1)\Delta) \geq E_j(2k\Delta)$. Soient deux instants t_1 et t_2 tels que $2(k-1)\Delta \leq t1 \leq 2k\Delta$ et $t_2 = t_1 + 2\Delta$. Supposons que les deux séquences soient identiques respectivement jusqu'à t_1 et t_2 avec $E_j(t_2) \geq E_j(t_1)$. Montrons qu'elles peuvent être identiques dans l'unité de temps suivante et que nous aurons aussi $E_j(t_2+1) \geq E_j(t_1+1)$. Entre t_1 et t_1+1, voici les différentes situations possibles :

- Le processeur est actif. Alors ayant d'avantage d'énergie dans le réservoir qu'à t_1, il peut aussi être actif entre t_2 et t_2+1. Et le niveau d'énergie dans le réservoir a diminué de la même quantité. D'où $E_j(t_2+1) = E_j(t_1+1) + E_j(2k\Delta) - E_j(2(k-1)\Delta)$. Et donc $E_j(t_2+1) \geq E_j(t_1+1)$.
- Le processeur est inactif. Il peut évidemment aussi être inactif entre t_2 et t_2+1. Trois situations sont toutefois à considérer du point de vue du réservoir :
  1. Le réservoir est plein à t_1 et de l'énergie est gaspillée entre t_1 et t_1+1 d'une quantité égale à P_r. La même quantité sera donc aussi gaspillée entre t_2 et t_2+1 et donc avec la même différence de quantité d'énergie dans le réservoir. Soit $E_j(t_2+1) = E_j(t_1+1) + E_j(2k\Delta) - E_j(2(k-1)\Delta)$. Et donc $E_j(t_2+1) \geq E_j(t_1+1)$.
  2. Le réservoir n'est pas plein ni à t_1, ni à t_2 . Dans ce cas, dans les deux séquences, le réservoir se remplit de la même quantité égale à Pr et nous avons aussi $E_j(t_2+1) = E_j(t_1+1) + E_j(2k\Delta) - E_j(2(k-1)\Delta)$. Et donc $E_j(t_2+1) \geq E_j(t_1+1)$.
  3. Le réservoir n'est pas plein à t_1 mais plein à t_2 . Soit $E_j(t_1) < B_j$ et $E_j(t_2) = B_j$. Nous avons donc $E_j(t_2+1) = B_j$ car l'énergie récupérée entre t_2 et t_2+1 est gaspillée. Et nous avons $E_j(t_1+1) \leq B_j$. Et donc $E_j(t_2+1) \geq E_j(t_1+1)$.

  Nous appliquons ce raisonnement par récurrence et montrons que $E_j(2(k+1)\Delta) \geq E_j(2k\Delta)$. Ceci implique que s'il existe un ordonnancement faisable pour τ sur $[0, 2\Delta]$ et si $E_j(2\Delta) \geq E_j(0)$, alors le même ordonnancement faisable existe pour sur $[2\Delta, 4\Delta]$ tel que $E_j(4\Delta) \geq E_j(2\Delta)$ et donc tel que $E_j(4\Delta) \geq E_j(0)$. Et par récurrence, il existe le même ordonnancement faisable sur $[2(k-1)\Delta, 2k\Delta]$ pour tout k, tel que $E_j(2k\Delta) \geq E_j(0)$. Donc, il existe un ordonnancement faisable qui est cyclique de période 2Δ. □

2.3 Dimensionnement suffisant du système du point de vue énergétique

L'objectif de cette partie est d'estimer la taille minimale $B_{j_{ef}}$ de la batterie qui permette de garantir les meilleures performances et ce, quel que soit l'algorithme choisi. Le dimensionnement est tel que : $\forall B_j \geq B_{j_{ef}}$, le taux de réussite reste le même.

En fait, nous pouvons prédir que $B_{j_{ef}}$ dépend de la charge énergétique U_{e_j} du processeur π_j qui elle-même est bornée pour que l'ensemble soit ordonnançable : $\forall j = 1..m, U_{e_j} \leq P_{r_j}$.

Nous énonçons donc le lemme suivant :

Lemme 4.1 *Pour un ensemble de tâches Γ_j (ayant un facteur d'utilisation énergétique U_{e_j}) s'exécutant sur un processeur π_j, la taille suffisante $B_{j_{ef}}$ de la batterie quelles que soient l'heuristique choisie et la source d'énergie considérée est égale à l'énergie consommée par l'ensemble Γ_j sur une hyperpériode $H(\Gamma_j)$, soit $B_{j_{ef}} = U_{e_j} \times H(\Gamma_j)$.*

Preuve: Nous nous plaçons dans le pire cas : le processeur est toujours actif sur $[0, H(\Gamma_j)]$ (l'exécution des tâches de l'ensemble Γ_j s'effectue sans aucun temps creux). Cela veut dire que la quantité d'énergie $U_e \times H(\Gamma_j)$ va être consommée dans la batterie de π_j entre $t = 0$ et $t = H(\Gamma_j)$. Ainsi, il faudra que cette quantité soit présente dans la batterie pour ne pas conduire à une famine énergétique. □

De plus, nous pouvons noter qu'à travers la taille de la batterie B_j (ou E_{max_j}) nous pouvons contrôler la charge énergétique maximale d'un processeur. En effet, réduire la taille de la batterie pousserait le processeur π_j à rejeter des tâches qui seraient assignées du coup sur le processeur suivant π_{j+1} impliquant ainsi une baisse de la charge énergétique de π_j.

3 Extension des heuristiques classiques de partitionnement à des systèmes autonomes énergétiquement

Les heuristiques de partitionnement classiques adaptées à des systèmes autonomes seront dénotées FF-AS (First Fit-Autonomous System heuristic), BF-AS (Best Fit-Autonomous System heuristic), et WF-AS (Worst Fit-Autonomous System heuristic).

3.1 Description algorithmique

La Figure 4.2 montre le pseudo-code d'un algorithme d'allocation *AA* adapté aux systèmes autonomes énergétiquement que l'on dénote AA-AS (AA-AS peut être FF-AS, BF-AS, WF-AS ou encore NF-AS). Pour chaque processeur π_j, Γ_j dénote les tâches parmi τ_1,..., τ_{i-1} qui ont déja été assignées au processeur π_j. Initialement, $\forall j = 1..m, \Gamma_j = \varnothing$.

Théorème 27 *Si l'algorithme AA-AS retourne SUCCES pour un ensemble de tâches τ à échéances contraintes, alors le partitionnement obtenu en utilisant l'heuristique AA est ordonnançable selon EDF.*

Preuve:

On observe que l'algorithme retourne SUCCES si et seulement s'il a assigné avec succès chaque tâche de τ aux différents processeurs. Avant l'assignation de la tâche τ_i, l'ensemble de tâches de chaque processeur est trivialement ordonnançable. Il résulte du Théorème 25 (ou 26 pour des tâches asynchrones) que tous les ensembles de tâches des processeurs restent ordonnançables après chaque attribution de tâche. Par conséquent, tous les ensembles de tâches des processeurs sont ordonnançables une fois que toutes les tâches de τ ont été assignées.□

4 Extension de KTS à des systèmes autonomes énergétiquement

4.1 Principe

L'approche KTS abstraction faite de l'énergie permet d'augmenter de manière significative les performances par rapport aux heuristiques de partitionnement classiques. Le choix d'étendre cette approche aux systèmes autonomes énergétiquement, désormais dénotée KTS-AA-AS va

Algorithme 3 Algorithme AA-AS

Entrées : $m, \tau = \{\tau_1, ..., \tau_n\}$

Sorties : SUCCES et $\{\Gamma_1, ..., \Gamma_m\}$ si τ est ordonnançable, ECHEC sinon.

Fonction : $FF - AS$

```
début
pour i = 1 → |τ| faire
    /* Tâche assignée selon l'heuristique AA, ici FF est utilisé */
    Ordo ← faux ;
    pour j = 1 → m faire
        si τ_i est temporellement-ordonnançable et énergétiquement-ordonnançable sur π_j alors
            /* τ_i est assignée au processeur π_j ; */
            Γ_j = Γ_j ∪ τ_i ;
            Ordo ← vrai ;
            break ;
        fin si
    fin pour
    si not Ordo alors
        retourner ECHEC ;
    fin si
fin pour
retourner SUCCES ;
fin
```

FIGURE 4.2 – Pseudo-code de AA-AS

permettre de diminuer le nombre d'ensembles de tâches qui n'étaient pas ordonnançables du fait du non respect des contraintes temporelles ou des contraintes d'énergie. Ce qui va permettre d'augmenter encore une fois les performances des heuristiques de base.

Le principe de fonctionnement de l'algorithme KTS-AA-AS reste similaire à celui décrit dans le chapitre 2. Seulement cette fois-ci, les tests d'ordonnnançabilité incluent le respect non seulement des contraintes temporelles mais aussi des contraintes d'énergie. Les tâches sont d'abord assignées aux différents processeurs selon une certaine heuristique de base associée à un critère de tri (basé, par exemple, sur l'utilisation ou la densité). Dès lors que l'allocation d'une tâche sur un processeur n'est plus possible (un rejet dû à (i) une violation d'échéance, (ii) à la présence d'une famine énergétique et/ou (iii) à un niveau d'énergie dans la batterie à la fin de la fenêtre de test en deça du niveau initial), la tâche rejetée va subir un fractionnement selon l'algorithme KTS, comme décrit dans la Figure 2.1 (page 41).

4.2 Description algorithmique

Tout comme l'algorithme de base de KTS abstraction faite de l'énergie, si aucun processeur n'est capable de recevoir une certaine tâche, nous ne pouvons rien conclure quant à l'ordonnançabilité de l'ensemble de tâches τ sur la plateforme multiprocesseur. La tâche τ_i est alors fractionnée en un sous-ensemble τ_i^{split} composé de deux tâches qui sont assignées à un sous-ensemble de processeurs de la plateforme à m processeurs. Le Lemme 4.2 affirme qu'en attribuant une tâche fractionnée de τ_i^{split} à un processeur π_j, l'ordonnançabilité des tâches confiées précédemment à l'ensemble des processeurs reste intacte.

Lemme 4.2 *Si l'ensemble de tâches précédemment assignées aux processeurs est ordonnançable (sur chaque processeur) et que par la suite l'algorithme KTS-AA-AS assigne une tâche τ_i^0 (respectivement τ_i^1) au processeur π_j (d'après le Théoreme 26), alors l'ensemble de tâches assignées*

à chaque processeur (y compris processeur π_j) reste ordonnançable.

Preuve: On observe que l'ordonnançabilité des processeurs autres que le processeur π_j n'est pas affectée par l'allocation de la tâche τ_i^0 (respectivement τ_i^1) au processeur π_j. Aussi, si l'ensemble de tâches précédemment assigné à π_j était faisable sur π_j avant l'allocation de τ_i^0 (respectivement τ_i^1) et que la condition du Théoreme 26 est satisfaite, alors l'ensemble de tâches sur π_j reste ordonnançable après l'ajout de τ_i^0 (respectivement τ_i^1).□

Le fonctionnement de KTS-AA-AS détermine, par des applications répétées du Lemme 4.2 au niveau de chaque tâche à assigner, l'ordonnançabilité de l'ensemble de tâches.

Théorème 28 *Considérant un ensemble de tâches périodiques τ, si l'algorithme de partitionnement KTS-AA-AS retourne SUCCES, alors le partitionnement établi est ordonnançable.*

Preuve: On observe que l'algorithme retourne SUCCES si et seulement s'il a assigné avec succès chaque tâche de τ aux différents processeurs. Avant l'assignation de la tâche τ_i, l'ensemble de tâches de chaque processeur est trivialement ordonnançable. Il résulte du Lemme 4.2 que tous les ensembles de tâches des processeurs restent ordonnançables après chaque attribution de tâche. Par conséquent, tous les ensembles de tâches des processeurs sont ordonnançables une fois que toutes les tâches de τ ont été assignées.□

5 Validation par simulation

L'objectif de ce paragraphe est de rapporter les différentes simulations effectuées dans le but d'évaluer les performances des heuristiques décrites précédemment, tout en analysant les critères qui permettent d'obtenir les meilleures performances. Pour cela, l'objectif est de maximiser le taux de réussite, c'est-à-dire la proportion d'ensembles de tâches générés qui s'exécutent dans le respect de leurs échéances tout en respectant les contraintes énergétiques du système. Les expérimentations en simulation tendent également à évaluer l'impact de la taille de la batterie sur ces différentes heuristiques et fournir ainsi un guide de choix au concepteur du système pour le dimensionnement des réservoirs de stockage d'énergie du système.

5.1 Environnement de simulation

L'environnement de simulation consiste en m ordonnanceurs associés aux différents processeurs constituant la plateforme. Le programme de simulation a été implémenté en langage Perl. L'architecture fonctionnelle du simulateur est représentée sur la Figure 4.3.

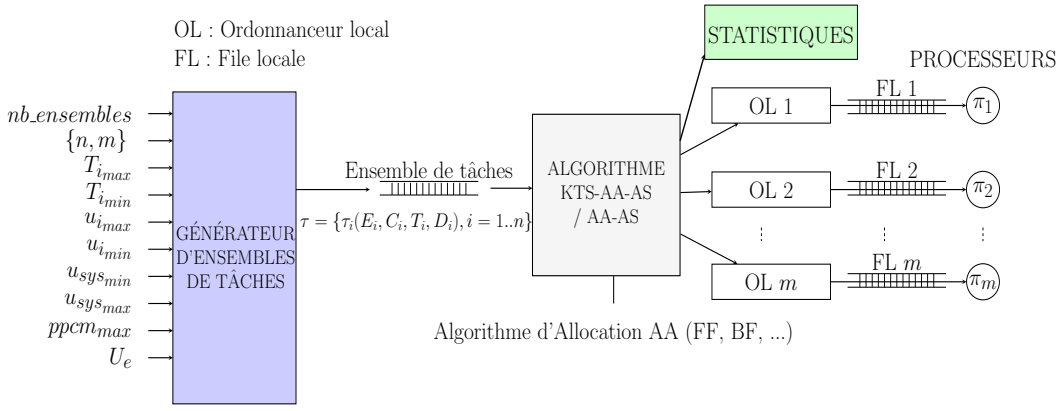


FIGURE 4.3 – Architecture fonctionnelle du simulateur

Le générateur d'ensembles de tâches périodiques a été conçu de manière à ce que les facteurs d'utilisation u_i des tâches de l'ensemble généré soient répartis de manière uniforme dans $[u_{i_{min}}, u_{i_{max}}] = [0.1, 1]$. Il accepte en entrée plusieurs paramètres : le nombre de processeurs m de la plateforme, le nombre de tâches n à partitionner, le nombre d'ensembles de tâches généré *nb_ensembles* pour chaque configuration, le ppcm maximum $ppcm_{max}$ des périodes des tâches choisies dans l'intervalle $[T_{i_{min}}, T_{i_{max}}] = [10, 200]$. La charge périodique totale du système (composé de m processeurs) u_{sys} varie dans $[u_{sys_{min}}, u_{sys_{max}}] = [0.6, 1]$.

En sortie, on obtient pour chaque valeur de u_{sys}, *nb_ensembles* ensembles de tâches périodiques synchrones $\tau = \{\tau_i(E_i, C_i, T_i, D_i), i = 1..n\}$. Les durées d'exécution des tâches sont générées à partir des facteurs d'utilisation et des périodes affectés aux tâches, $C_i = u_i \times T_i$. Les consommations énergétiques pire-cas E_i des tâches sont choisies aléatoirement de manière à ce que leur somme soit égale au facteur d'utilisation énergétique totale U_e fourni en entrée.

Dans notre méthode de génération de tâches, nous considérons des tâches à échéances contraintes et donc D_i est choisie aléatoirement dans $[C_i, T_i]$.
Après la génération des différents ensembles de tâches, le simulateur tente de partitionner, pour chaque valeur de u_{sys}, les *nb_ensembles* ensembles de tâches et sort la proportion d'ensembles de tâches jugés ordonnançables appelée le *taux de réussite* (*success ratio* en anglais). Le taux de réussite est le critère de performance qui permettra de comparer les différentes stratégies entre elles.

Dans toutes nos simulations, nous générons 300 ensembles de 20 tâches à échéances contraintes ($D \leq T$) possédant différents niveaux de criticité d'énergie R_e. Le nombre de processeurs est fixé à 8.

5.2 Évaluation des heuristiques adaptées FF-AS, BF-AS et WF-AS

Dans un premier temps, nous cherchons à évaluer les performances des heuristiques de base adaptées à des systèmes autonomes énergétiquement. En particulier, nous étudions l'influence du critère de tri des différentes tâches de l'ensemble ainsi que la taille de la batterie sur le taux de réussite observé. Par la suite, nous utiliserons ces résultats comme référence pour l'évaluation de KTS-AA-AS.

5.2.1 Effet du critère de tri

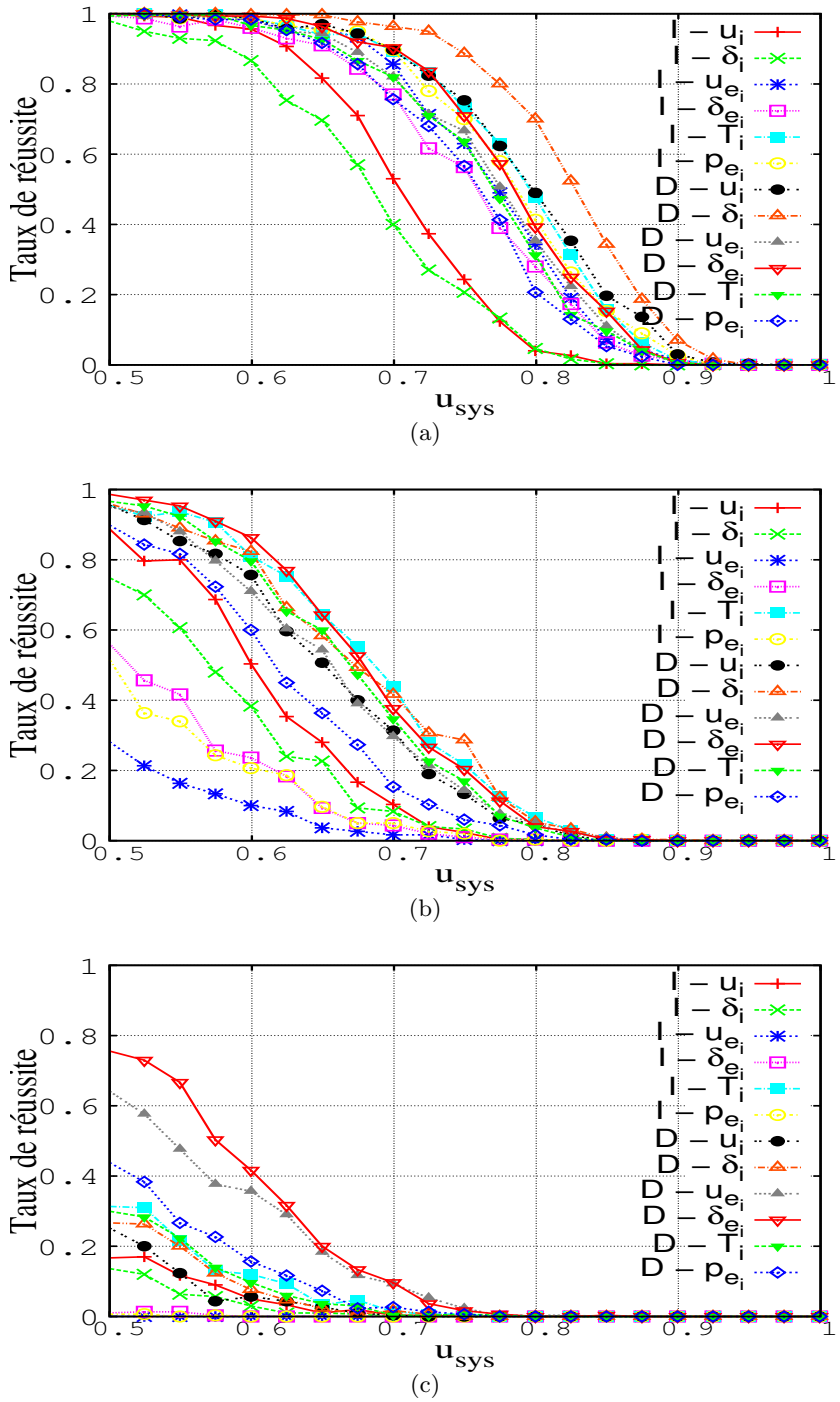
Les Figures 4.4, 4.5 et 4.6 montrent l'effet du tri des tâches à échéances contraintes selon un critère donné en amont de la phase de partitionnement pour les différentes heuristiques BF-AS, FF-AS et WF-AS pour différentes contraintes d'énergie $R_e = \{0.3, 0.75, 0.9\}$. Sur ces figures, D signifie décroissant (*decreasing*) et I signifie croissant (*increasing*).

On remarque que pour les systèmes faiblement contraints énergétiquement, la courbe du critère de densité décroissante, $D{-}\delta_i$, domine toutes les autres courbes et offre donc les meilleures performances et ce quelle que soit l'heuristique considérée (voir les Figures 4.4(a), 4.5(a) et 4.6(a)).

WF-AS fait exception : $D - \delta_i$ reste invariablement le meilleur critère de tri pour avoir de très bonnes performances quelle que soit la contrainte énergétique (voir Figure 4.6(b)).

Cependant pour BF-AS et FF-AS, à mesure que le système devient plus fortement contraint énergétiquement (cas extrême $R_e = 0.9$), le tri des tâches par leur densité énergétique et leur utilisation énergétique de manière décroissante $D - \delta_{e_i}$ et $D - u_{e_i}$ offrent les meilleures performances (voir les Figures 4.4(c), 4.5(c) et 4.6(c)) pour les trois heuristiques. Il est important de noter que, pour des ensembles de tâches comportant des tâches dont les facteurs d'utilisation énergétique peuvent être élevés, le fait d'assigner d'abord les tâches possédant les plus grands facteurs d'utilisation énergétique réduit la probabilité de ne pas pouvoir assigner ces tâches ultérieurement pour cause de non-respect des contraintes énergétiques.

Enfin, pour des systèmes moyennement contraints énergétiquement, selon la prédominance d'une contrainte (temporelle ou énergétique) sur une autre, les performances de critères vont légèrement varier et donc on peut s'attendre à voir des des résultats assez proches les uns des autres. En effet, pour BF-AS et FF-AS, les courbes $D - \delta_{e_i}$, $D - \delta_i$ et $I - T_i$ se chevauchent et offrent par ailleurs les meilleures performances (voir les Figures 4.4(b), 4.5(b)). Il est clair que pour des systèmes consommateurs en énergie, le choix d'assigner en premier les tâches ayant un facteur d'utilisation d'énergie élevé réduit la possibilité d'avoir par la suite un rejet de tâches dû aux contraintes d'énergie. Les critères de tri associés aux contraintes temporelles des tâches ne sont pour autant pas exclues. En particulier, pour des systèmes à taux d'utilisation important, les critères $I - T_i$ et $D - \delta_i$ donnent des performances similaires à celles observées pour les critères $D - \delta_{e_i}$ et $D - \delta_i$.
Nous effectuerons donc dans la suite : (i) un tri des tâches par ordre décroissant de leur densité $D - \delta_i$ pour des systèmes faiblement ou moyennement contraints, (ii) un tri des tâches par ordre décroissant de leur densité $D - \delta_{e_i}$ pour des systèmes fortement contraints



(a)

(b)

(c)

FIGURE 4.4 – Performances de BF-AS selon différents niveaux de criticité d'énergie : (a) $R_e = 0.3$, (b) $R_e = 0.75$, (c) $R_e = 0.9$

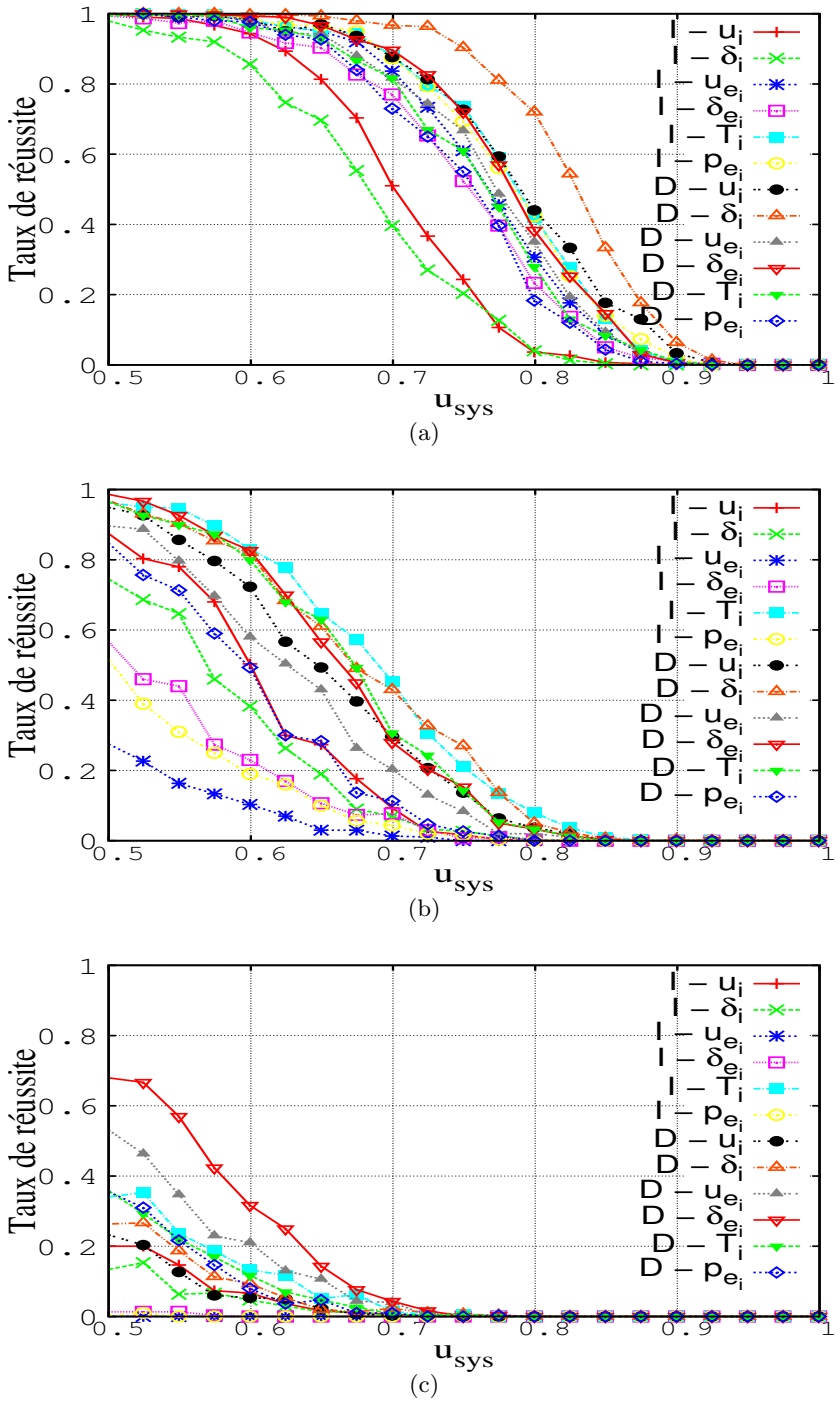


FIGURE 4.5 – Performances de FF-AS selon différents niveaux de criticité d'énergie : (a) $R_e = 0.3$, (b) $R_e = 0.75$, (c) $R_e = 0.9$

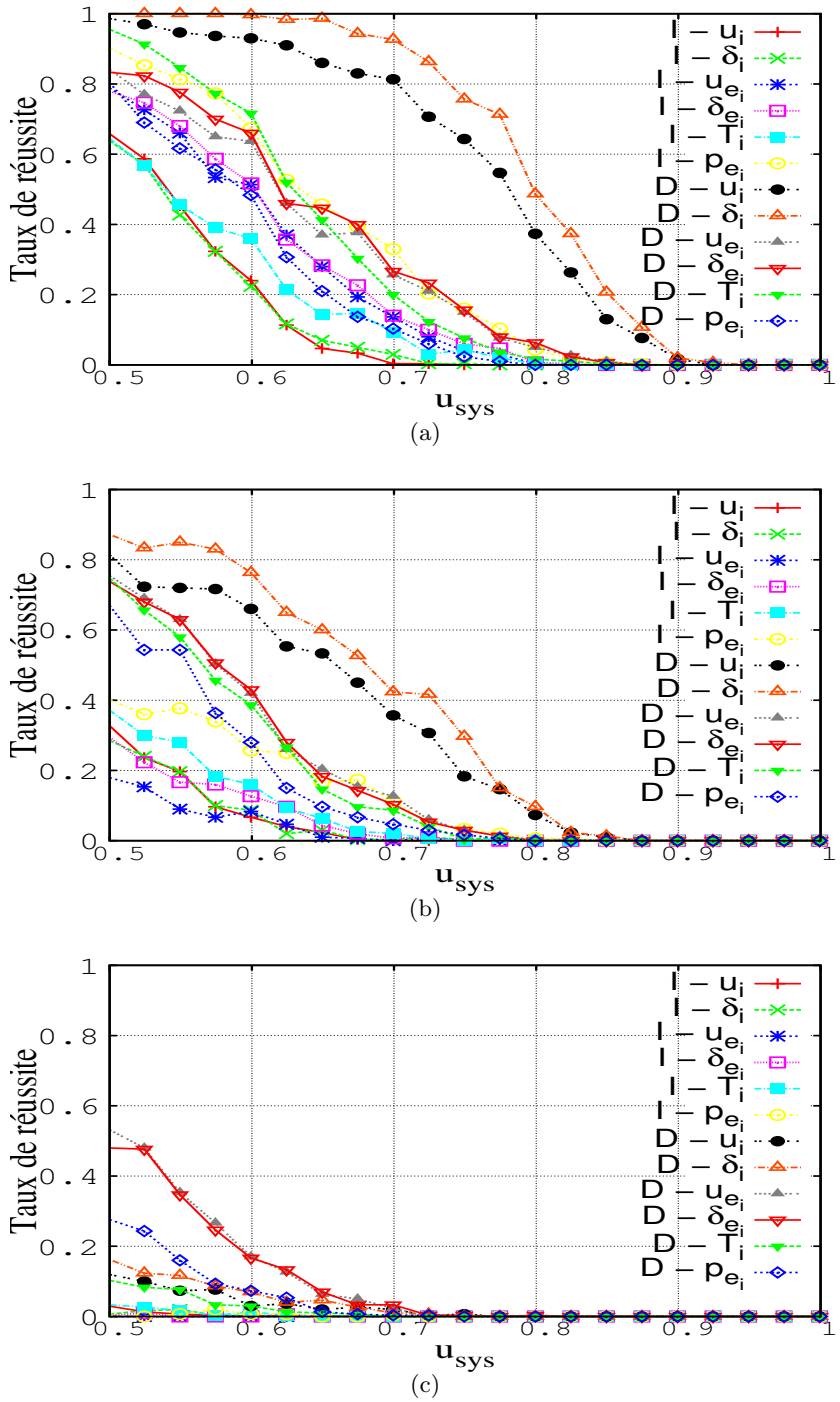
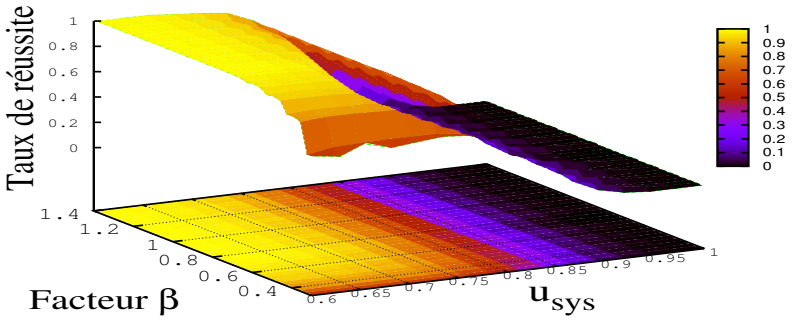


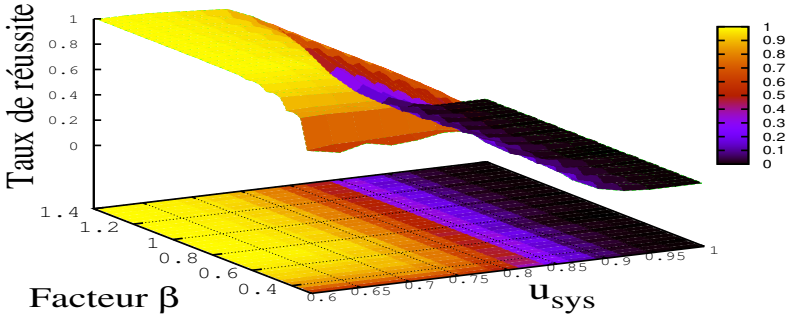
FIGURE 4.6 – Performances de WF-AS selon différents niveaux de criticité d'énergie : (a) $R_e = 0.3$, (b) $R_e = 0.75$, (c) $R_e = 0.9$

5.2.2 Effet de la taille de la batterie

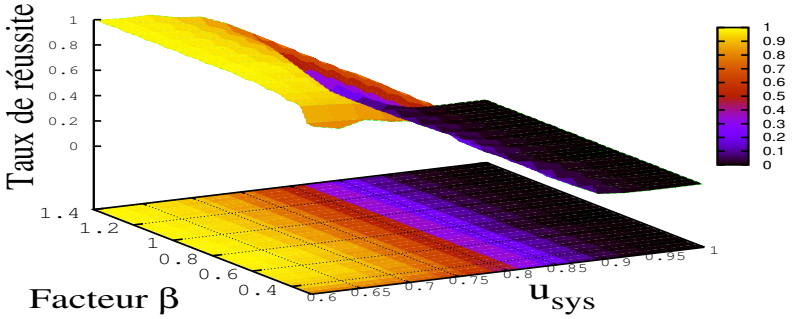
Les Figures 4.7, 4.8 et 4.9 illustrent les performances des différentes heuristiques lorsque la taille B_j de la batterie de chaque processeur varie. Pour cela, nous observons le taux de réussite des 300 ensembles de tâches pour chaque valeur de R_e et ce, avec différentes tailles de batteries toutes reliées à un facteur près β à la consommation énergétique du processeur π_j pendant une hyperpériode, soit $B_j = \beta \times (U_{e_j} \times H(\Gamma_j))$.



(a)

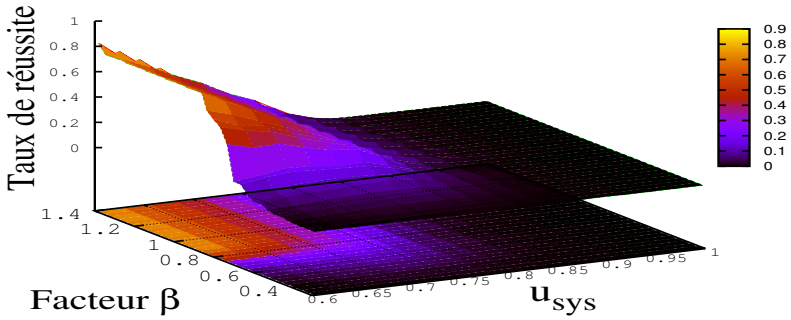


(b)

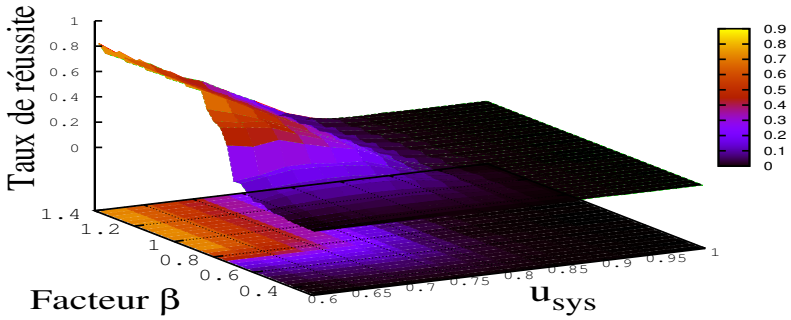


(c)

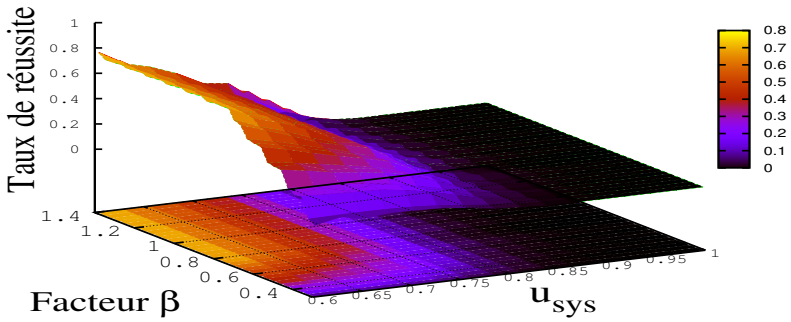
FIGURE 4.7 – Effet de la taille de la batterie B_j pour R_e = 0.3 sur le taux de réussite de : (a) BF-AS, (b) FF-AS, (c) WF-AS



(a)

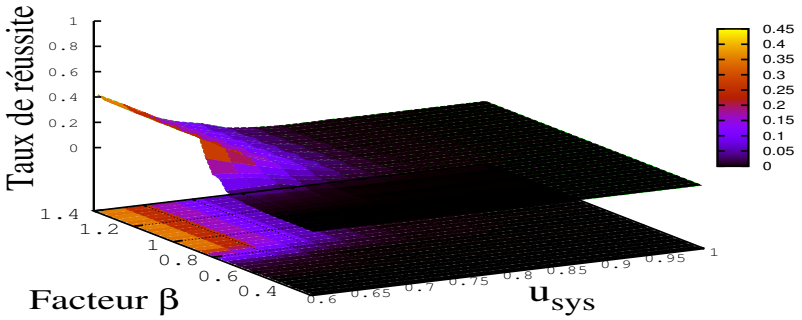


(b)

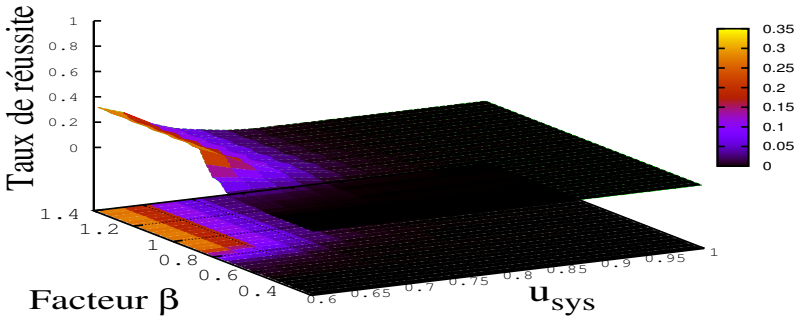


(c)

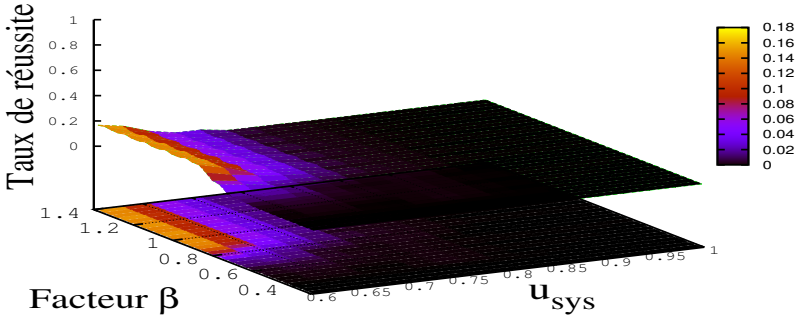
FIGURE 4.8 – Effet de la taille de la batterie B_j pour $R_e = 0.75$ sur le taux de réussite de : (a) BF-AS, (b) FF-AS, (c) WF-AS



(a)



(b)



(c)

FIGURE 4.9 – Effet de la taille de la batterie B_j pour R_e = 0.9 sur le taux de réussite de : (a) BF-AS, (b) FF-AS, (c) WF-AS

Comme prévu, les Figures 4.7, 4.8 et 4.9 montrent que quels que soient l'heuristique et le niveau de criticité d'énergie R_e, à partir d'un facteur β = 1, le taux de réussite cesse d'augmenter.

5.3 Évaluation de l'approche $KTS - AA - AS$

Nous cherchons à évaluer KTS-AA-AS pour des systèmes possédant différents niveaux de criticité énergétique et voir dans quelle situation cette approche est la plus avantageuse.

5.3.1 Effet de la profondeur limite K

Les Figures 4.10, 4.11, et 4.12 illustrent le taux de réussite de AA-AS et KTS-AA-AS pour différents algorithmes d'allocation (AA = { BF, FF, WF}). Nous considérons toujours des ensembles de tâches à échéances contraintes avec différents niveaux de criticité d'énergie : $R_e = \{0.3; 0.75; R_e = 0.9\}s$.

Les résultats des Figures 4.10, 4.11, et 4.12 montrent que KTS adapté offre toujours de meilleures performances et ce, quels que soient l'algorithme d'allocation et le niveau de criticité choisis. Il permet d'augmenter progressivement le taux de réussite à mesure que l'on augmente la profondeur limite K jusqu'à une certaine limite $K = \log_2(m)$ (voir page 45). De plus, nous notons que l'écart de performances entre KTS-AA-AS et les heuristiques classiques est le plus grand dans le cas où le système est moyennement contraint énergétiquement. Nous observons par exemple dans la Figure 4.11, pour $u_{sys} = 0.775$:

- lorsque $R_e = 0.3$, KTS-FF-AS (K=3) améliore de 4.3% le taux de réussite par rapport à FF-AS.
- $R_e = 0.75$, KTS-FF-AS (K=3) améliore de 13.3% le taux de réussite par rapport à FF-AS.
- $R_e = 0.9$, KTS-FF-AS (K=3) améliore de 0.3% le taux de réussite par rapport à FF-AS.

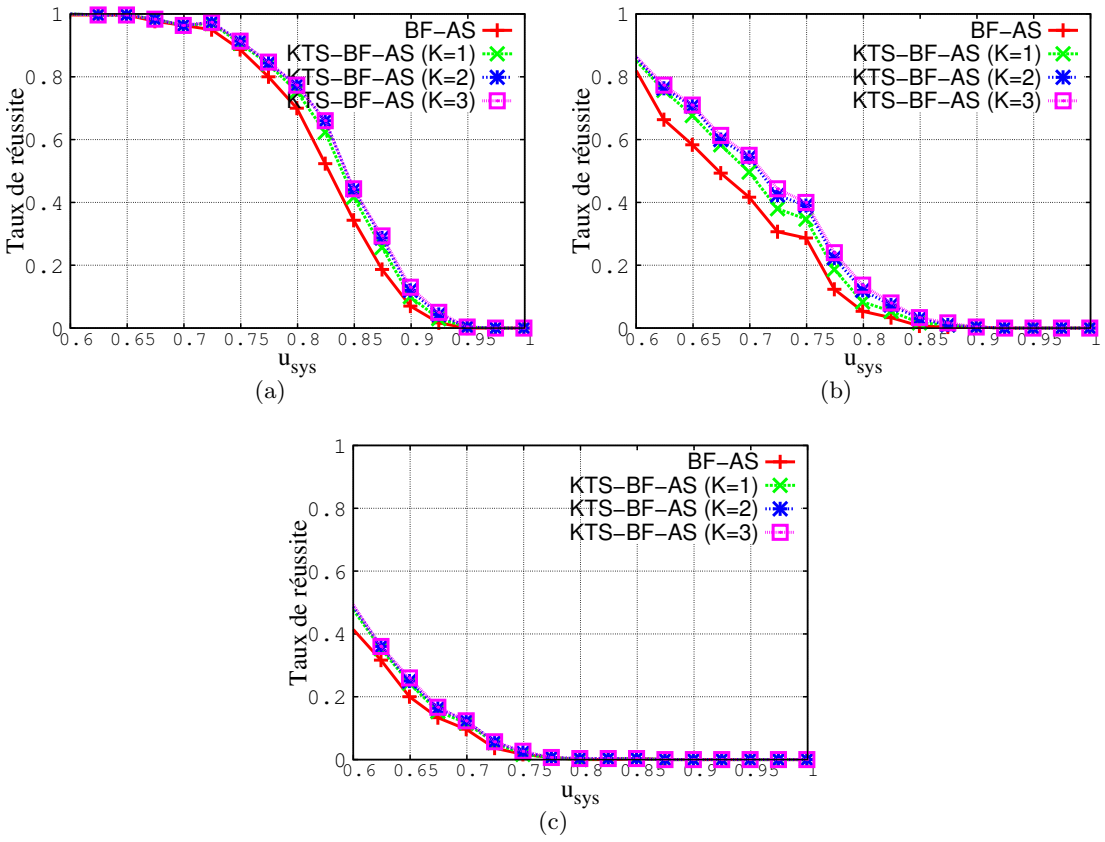


FIGURE 4.10 – Effet de la profondeur limite K pour KTS-BF-AS selon différents niveaux de criticité d'énergie R_e : (a) $R_e = 0.3$, (b) $R_e = 0.3$, (c) $R_e = 0.9$

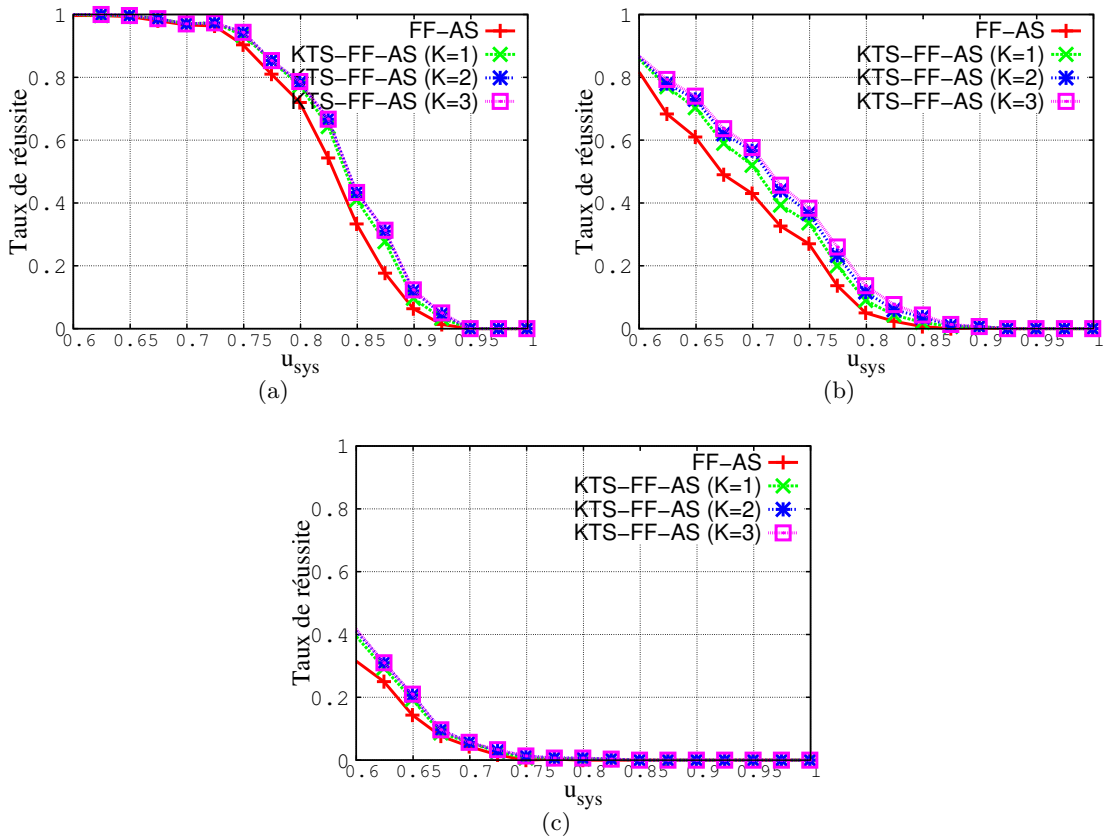


FIGURE 4.11 – Effet de la profondeur limite K pour KTS-FF-AS selon différents niveaux de criticité d'énergie R_e : (a) $R_e = 0.3$, (b) $R_e = 0.3$, (c) $R_e = 0.9$

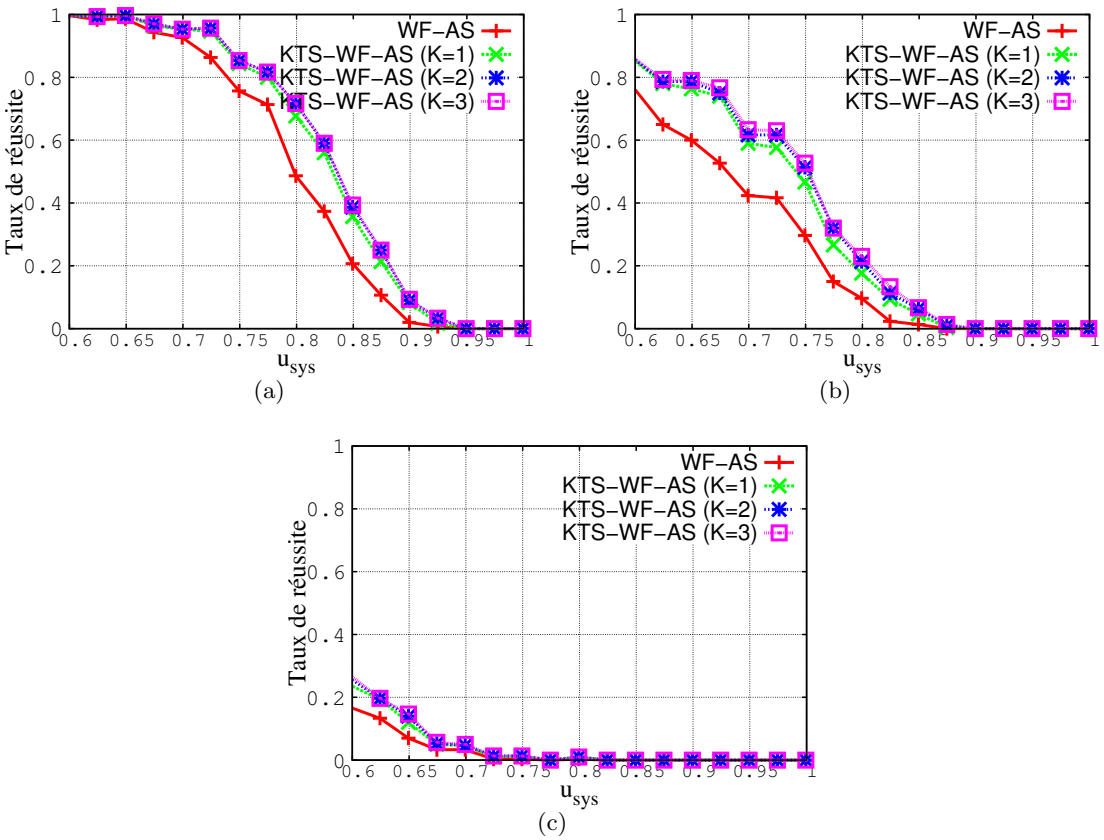


FIGURE 4.12 – Effet de la profondeur limite K pour KTS-WF-AS selon différents niveaux de criticité R_e : (a) $R_e = 0.3$, (b) $R_e = 0.3$, (c) $R_e = 0.9$

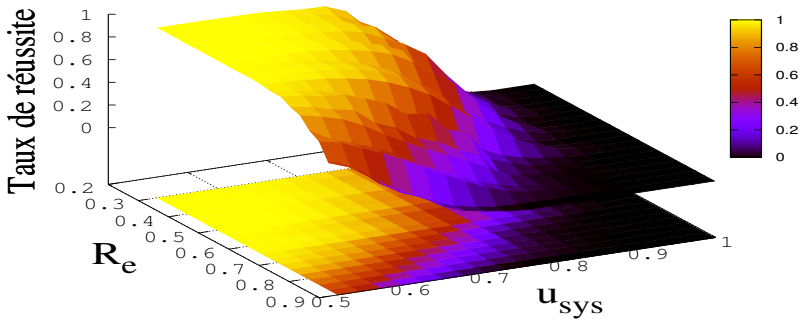
5.3.2 Effet du ratio d'énergie R_e

La Figure 4.13 illustre les performances de l'approche de type task splitting KTS-AA-AS vis-à-vis des heuristiques de base AA-AS sur une plateforme à 8-processeurs pour des tâches à échéances contraintes en faisant varier dans le niveau de criticité d'énergie R_e. Les résultats quelle que soit l'heuristique choisie sont similaires. Nous choisissons d'afficher ceux correspondant à l'heuristique de bin-packing FF.

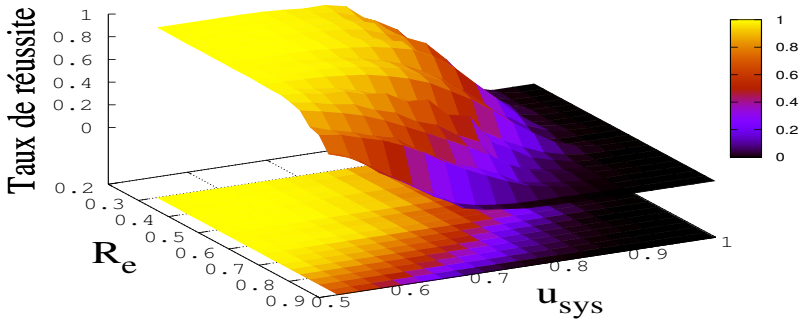
Comme prévu, à mesure que R_e augmente, AA-AS tout comme KTS-AA-AS peinent à déterminer un partitionnement ordonnançable, ce qui se traduit par une diminution du taux de réussite observé. Par exemple, lorsque l'utilisation du système $u_{sys} = 0.7$, parmi tous les ensembles de tâches générés,

- FF-AS en ordonnance (i) 96.7% lorsque $R_e = 0.3$, (ii) 92.3% lorsque $R_e = 0.6$ et (iii) 31.7% lorsque $R_e = 0.8$.
- KTS-FF-AS avec $K = 3$ en ordonnance (i) 0.33% de plus (que FF-AS) lorsque $R_e = 0.3$, (ii) 5.7% de plus lorsque $R_e = 0.6$ et (iii) 9% de plus lorsque $R_e = 0.8$.

Notons cependant que KTS-AA-AS (K=3) est plus lentement affecté par les changements de R_e : la courbe du taux de réussite décroit moins vite pour KTS-FF-AS (K=3) que pour FF-AS. Nous pouvons en conclure que KTS-AA-AS est une solution d'autant plus intéressante que le système est fortement contraint énergétiquement.



(a)



(b)

FIGURE 4.13 – Effet du niveau de criticité d'énergie R_e : (a) FF-AS , (b) KTS-FF-AS

6 Conclusion

Dans ce chapitre, nous nous sommes intéressés au problème du partitionnement sur des plateformes multiprocesseur autonomes énergétiquement. Nous avons abordé le problème de la répartition d'un ensemble de tâches sur un système multiprocesseur alimenté par une source d'énergie renouvelable. Une analyse d'ordonnançabilité pour EDF a été présentée pour prendre en compte à la fois les contraintes temporelles et d'énergie pour des tâches périodiques synchrones. Nous avons ensuite étendu cette analyse pour un modèle de tâches périodiques asynchrones.

Nous avons également déterminé quel était le dimensionnement suffisant de tels systèmes estimant la capacité minimale des différentes batteries permettant d'obtenir le meilleur taux de réussite et ce, quels que soient le niveau de contrainte énergétique du système et/ou l'heuristique considérés. Celle-ci est égale à l'énergie consommée par l'ensemble Γ_j assigné au processeur π_j sur une hyperpériode $H(\Gamma_j)$.

L'évaluation ensuite sur une large gamme d'ensembles de tâches indique que pour des systèmes fortement contraints énergétiquement, le tri des tâches par leur densité énergétique décroissante améliore les performances. De plus, comme prévu intuitivement, le taux de réussite augmente à mesure que le niveau de criticité d'énergie R_e diminue.

Par la suite, nous avons étendu et évalué la performance de l'approche KTS adaptée aux systèmes autonomes énergétiquement (KTS-AA-AS). Comparativement aux heuristiques de base adaptées AA-AS, à mesure que l'on augmente la profondeur limite K, la performance

de KTS-AA-AS s'améliore. L'écart de performance le plus visible est observé dans le cas de systèmes moyennement contraints énergétiquement.

Le chapitre suivant porte sur l'extension de ces algorithmes à la catégorie de tâches à contraintes fermes permettant à certains jobs d'être ignorés ou abandonnés au cours de leur exécution sans conséquences graves sur le système. Le but dans ce cas est de pouvoir faire face à des cas de surcharge temporelle et/ou de famine énergétique.

Chapitre 5

Contribution au partitionnement sous contraintes d'énergie et de QoS

Ce chapitre est consacré au problème relatif au partitionnement d'un ensemble de tâches temps réel périodiques à contraintes fermes sur une plateforme multiprocesseur homogène autonome du point de vue énergétique. Nous formalisons le problème et les conditions d'ordonnançabilité présents pour le modèle de tâches et la plateforme considérés. Nous adaptons ensuite les heuristiques de partitionnement classiques pour répartir des tâches temps réel fermes en respectant les contraintes énergétiques et temporelles sur chaque processeur en utilisant Earliest Deadline First (EDF) comme politique d'ordonnancement. Dans un deuxième temps, nous proposons une extension de l'algorithme KTS pour la répartiton d'ensemble de tâches fermes sur des systèmes multiprocesseur homogène autonome du point de vue énergétique. Ces différentes stratégies sont validées au travers de différentes simulations. En particulier, nous étudions dans quelle mesure la flexibilité apportée par les pertes autorisées au niveau des tâches permet de mieux faire face aux situations de surcharge de traitement et/ou de surcharge énergétique.

1 Système considéré

Nous nous intéressons dans cette partie à la proposition de solutions de répartition de tâches temps réel tolérantes aux pertes sur des systèmes nomades qui par nature doivent être autonomes du point de vue énergétique. Nous considérons dans ce chapitre, tout comme dans le chapitre 4, des systèmes embarqués qui fonctionnent grâce à des réservoirs d'énergie, qui se rechargent continuement au cours du temps à partir d'une source d'énergie renouvelable. Toutefois, les techniques développées permettant d'assurer un fonctionnement à l'infini dans le respect des contraintes temporelles des seuls jobs de tâches obligatoires, en utilisant uniquement l'énergie disponible dans le réservoir et sans jamais en manquer.

1.1 Modèle de tâches

Nous considérons un ensemble de n tâches fermes indépendantes, périodiques et préemptibles $\tau = \{\tau_1, \tau_2, ..., \tau_n\}$. Chaque tâche $\tau_i\,(O_i, E_i, C_i, T_i, D_i, s_i)$ est définie par :

- O_i, l'offset de τ_i ;
- E_i, sa consommation énergétique pire-cas (WCEC : Worst Case Energy Consumption), exprimée en joules ;
- C_i, son temps d'exécution pire-cas ;
- T_i, sa période ;
- D_i, son délai critique ou échéance relative ;
- s_i, le paramètre de pertes de τ_i conformément au modèle Skip-Over [KS95].

Nous nous intéressons au modèle de tâches profondément rouge à échéances contraintes ($D_i \leq T_i$).

La consommation énergétique E_i de τ_i correspond donc à la plus grande quantité d'énergie qu'elle peut consommer en s'exécutant sur un processeur. Le facteur d'utilisation ou charge processeur de la tâche τ_i, dénoté u_i, correspond au taux d'activité du processeur dédié à l'exécution des jobs successifs de la tâche : $u_i = \frac{C_i}{T_i}$. Le facteur d'utilisation d'un système composé de m processeurs et n tâches périodiques est défini comme suit :

$$u_{sys} = \frac{1}{m} \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{5.1}$$

Nous définissons une nouvelle notion qui est le facteur d'utilisation équivalent d'un système composé de m processeurs et n tâches périodiques fermes qui se définit comme suit :

$$u^*_{sys} = \frac{1}{m} \sum_{i=1}^{n} \frac{C_i}{T_i} . \frac{s_i - 1}{s_i} \tag{5.2}$$

Par ailleurs, une tâche τ_i est également caractérisée par : (i) son facteur d'utilisation énergétique $u_{e_i} = \frac{E_i}{T_i}$, (ii) son facteur d'utilisation énergétique équivalent (considérant les pertes autorisées) $u^*_{e_i} = \frac{E_i}{T_i} . \frac{s_i - 1}{s_i}$, (iii) sa densité énergétique $\delta_{e_i} = \frac{E_i}{\min(D_i, T_i)}$ et (iv) sa densité énergétique équivalente (considérant les pertes autorisées) $\delta^*_{e_i} = \frac{E_i}{\min(D_i, T_i)} . \frac{s_i - 1}{s_i}$. Le facteur d'utilisation énergétique d'un ensemble de tâches τ est alors défini comme suit : $U_e = \sum_{i=1}^{n} u_{e_i}$. Le facteur d'utilisation énergétique équivalent d'un ensemble de tâches τ s'exprime quant à lui de la manière suivante : $U^*_e = \sum_{i=1}^{n} u^*_{e_i}$.

Chaque processeur π_j est destiné à l'exécution du sous-ensemble de tâches qui lui sont assignées, soit Γ_j. Nous noterons $H(\Gamma_j) = PPCM(T_i)$ l'hyperpériode de Γ_j et $H^*(\Gamma_j) = PPCM(s_i T_i)$ l'hyperpériode équivalente de Γ_j (considérant les pertes autorisées). Les surcoûts en temps et en énergie, occasionnés par les éventuelles préemptions, sont supposés nuls.

Le modèle du système est illustré sur la Figure 5.1. Nous considérons une plateforme multiprocesseur composée de m processeurs identiques (π_j dénote le j-ième processeur de la plateforme) où chaque processeur π_j consomme de l'énergie de façon indépendante depuis son propre réservoir d'énergie de capacité B_j. À un instant t, chaque récupérateur d'énergie récupère l'énergie à partir d'une source d'énergie et la convertit en énergie électrique stockée dans son réservoir d'énergie de capacité B_j de manière à ce qu'à aucun instant, l'énergie stockée dans le réservoir de π_j, dénotée $E_j(t)$, ne dépasse sa capacité. Si celle-ci atteint la capacité du réservoir, l'énergie récupérée entrante ne peut être stockée dans le réservoir et est donc rejetée. De même, le réservoir est considéré complètement vide lorsqu'il atteint un niveau minimum $E_{min} = 0$.

Les stratégies de partitionnement, ici, se feront sur la base d'une garantie d'exécution des jobs rouges des tâches. L'ordonnanceur local quant à lui pourra tenter ensuite en-ligne et ce, en fonction de l'état du système, l'exécution des jobs bleus des tâches.

1.2 Cas de surcharge considérés

- **Surcharge de traitement** : Nous disons qu'un système est en surcharge de traitement lorsque l'utilisation du système dépasse 100% ($u_{sys} > 1$) pour pouvoir respecter les échéances des tâches indépendemment de leurs besoins en énergie.
- **Surcharge énergétique** : Nous disons qu'un ensemble de tâches est en surcharge énergétique lorsqu'il a des besoins en énergie trop élevés par rapport à l'énergie reçue de l'environnement pour pouvoir respecter les échéances des tâches indépendamment de la quantité de traitement requise, soit $R_e > 1$.

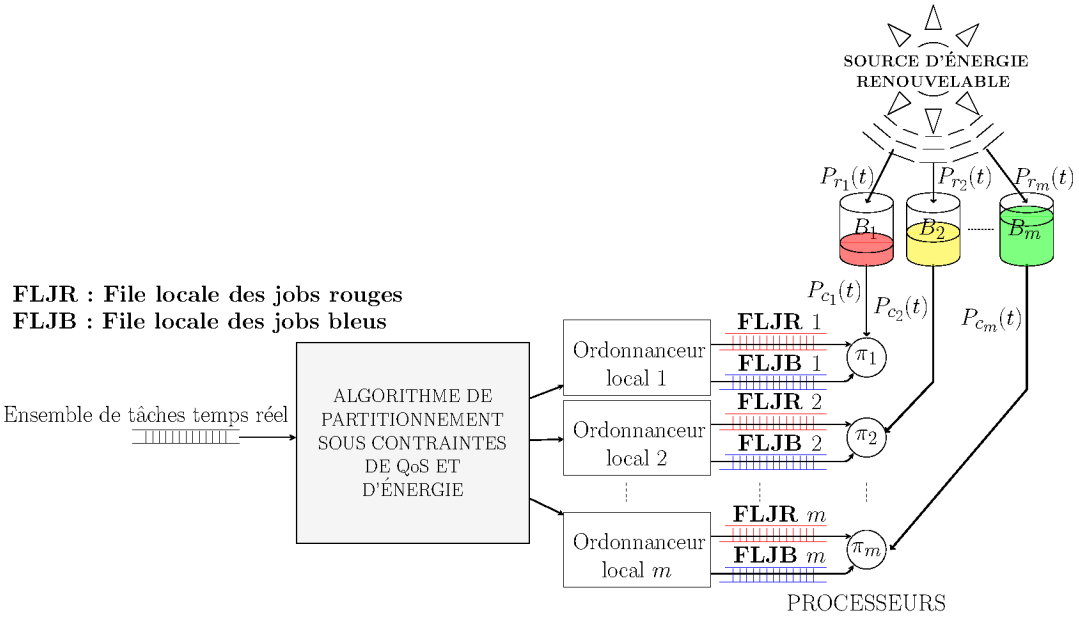


FIGURE 5.1 – Illustration du système autonome considéré

2 Le partitionnement sous contraintes de QoS et d'énergie

2.1 Formulation du problème

Le problème soulevé ici consiste à trouver un partitionnement *valide* c'est-à-dire satisfaisant à la fois les contraintes temporelles et les contraintes énergétiques liées aux jobs rouges des tâches uniquement. Celui-ci permettra de déterminer la répartition d'un ensemble des tâches fermes $\tau = \{\tau_1, \tau_2, ...\tau_n\}$ sur les différents processeurs de la plateforme $\Pi = \{\pi_1, ..., \pi_m\}$, de manière à ce que la configuration globale soit ordonnançable. Plus formellement, il s'agit de partitionner l'ensemble des n tâches en m sous-ensembles disjoints $\Gamma_1, \Gamma_2, ..., \Gamma_m$ (avec $\bigcup \Gamma_j = \tau$) et d'ordonnancer ensuite chaque sous-ensemble Γ_j sur le processeur π_j avec la politique d'ordonnancement monoprocesseur EDF en respectant :

- les contraintes d'ordre temporel : Sur chaque processeur π_j, tous les jobs rouges de tâches de Γ_j doivent s'exécuter avant leurs échéances.
- les contraintes d'ordre énergétique : Chaque processeur π_j ne doit, à aucun instant, manquer d'énergie pour exécuter l'ensemble des jobs rouges de tâches de Γ_j assignées à un processeur π_j.

2.2 Tests d'ordonnançabilité

En faisant abstraction de la contrainte énergétique, nous avons proposé dans le chapitre 3 une condition exacte (voir Théorèmes 19 et 20 page 60) pour respectivement des ensembles de tâches périodiques synchrones et asynchrones.
En rajoutant **la dimension énergétique** au problème, nous étendons les conditions d'ordonnançabilité pour garantir les contraintes temporelles et d'énergie :

Théorème 29 *Un ensemble Γ_j de n tâches périodiques synchrones à échéances contraintes possédant un facteur d'utilisation énergétique U_{e_j} est ordonnançable sur un processeur π_j considérant un profil énergétique constant si et seulement si :*

1. $Load_j^*(\Gamma_j) \leq 1$.
2. $\forall 0 \leq t \leq H^*(\Gamma_j), E_j^*(t) \geq 0$.

3. $E_j^*(H^*(\Gamma_j)) \geq E_j^*(0)$.

avec

$$E_j^*(t) = \min(B_j, E_j(t-1) + P_{r_j}(t-1,t) - E_{c_j}^*(t-1,t)) \quad (5.3)$$

où $E_{c_j}^*(t-1,t)$ correspond à l'énergie requise entre $t-1$ et t pour l'exécution des jobs rouges uniquement associés aux tâches assignées au processeur π_j.

Preuve:

Seulement si.

Il paraît évident que si l'une des conditions n'est pas respectée, cela conduira à la non-ordonnançabilité de l'ensemble de tâches :

- si la condition 1 n'est pas respectée, un (ou plusieurs) jobs rouges va (vont) manquer son (leur) échéance comme la charge lié à leur exécution dépasse la capacité du processeur à les exécuter ;
- si la condition 2 n'est pas respectée, à un moment le processeur va manquer d'énergie pour exécuter les jobs ;
- si la condition 3 n'est pas respectée, l'état du réservoir va progressivement diminuer d'une hyperpériode sur l'autre. Comme la charge énergétique est la même sur chaque hyperpériode, le niveau d'énergie dans le réservoir va progressivement chuter : $E_j^*(0) \geq E_j^*(H^*(\Gamma_j)) \geq E_j^*(2H^*(\Gamma_j)) ... \geq E_j^*(kH^*(\Gamma_j))$, ce qui entraînera fatalement une famine énergétique.

Si.

L'ensemble de tâches est ordonnançable sur $[(k-1)H^*(\Gamma_j), kH^*(\Gamma_j)]$, donc tous les jobs rouges de tâches réveillés dans cet intervalle ont terminé leur exécution avant échéance (condition 1. du Théoreme 19) et ce, sans famine énergétique ($E_j^*(t) \geq 0$). À l'instant $kH^*(\Gamma_j)$, le système, du point de vue de l'activité du processeur se retrouve dans le même état qu'à $(k-1)H^*(\Gamma_j)$ (cyclicité de période $H^*(\Gamma_j)$). Du point de vue énergétique, le système se retrouve à l'instant $kH^*(\Gamma_j)$ dans une situation plus favorable qu'à $(k-1)H^*(\Gamma_j)$ avec un surplus d'énergie dans le réservoir à l'instant $kH^*(\Gamma_j)$, égal à $E_j^*(kH^*(\Gamma_j)) - E_j^*((k+1)H^*(\Gamma_j))$. Par conséquent, l'ensemble sera aussi ordonnançable sur $[kH^*(\Gamma_j), (k+1)H^*(\Gamma_j)]$. Montrons maintenant que nous pouvons appliquer aux jobs rouges réveillés sur $[kH^*(\Gamma_j), (k+1)H^*(\Gamma_j)]$ le même ordonnancement que celui appliqué aux jobs rouges réveillés sur $[(k-1)H^*(\Gamma_j), kH^*(\Gamma_j)]$ et nous aurons alors nécessairement $E_j^*((k+1)H^*(\Gamma_j)) \geq E_j^*(kH^*(\Gamma_j))$. Soient deux instants t_1 et t_2 tels que $(k-1)H^*(\Gamma_j) \leq t1 \leq kH^*(\Gamma_j)$ et $t_2 = t_1 + H^*(\Gamma_j)$. Supposons que les deux séquences soient identiques respectivement jusqu'à t_1 et t_2 avec $E_j^*(t_2) \geq E_j^*(t_1)$. Montrons qu'elles peuvent être identiques dans l'unité de temps suivante et que nous aurons aussi $E_j^*(t_2+1) \geq E_j^*(t_1+1)$. Entre t_1 et t_1+1, voici les différentes situations possibles :

- Le processeur est actif. Alors ayant davantage d'énergie dans le réservoir qu'à t_1, il peut aussi être actif entre t_2 et t_2+1. Et le niveau d'énergie dans le réservoir a diminué de la même quantité. D'où $E_j^*(t_2+1) = E_j^*(t_1+1) + E_j^*(kH^*(\Gamma_j)) - E_j^*((k-1)H^*(\Gamma_j))$. Et donc $E_j^*(t_2+1) \geq E_j^*(t_1+1)$.
- Le processeur est inactif. Il peut évidemment aussi être inactif entre t_2 et t_2+1. Trois situations sont toutefois à considérer du point de vue du réservoir :
  1. Le réservoir est plein à t_1 et de l'énergie est gaspillée entre t_1 et t_1+1 d'une quantité égale à Pr . La même quantité sera donc aussi gaspillée entre t_2 et t_2+1 et donc avec la même différence de quantité d'énergie dans le réservoir. Soit $E_j^*(t_2+1) = E_j^*(t_1+1) + E_j^*(kH^*(\Gamma_j)) - E_j^*((k-1)H^*(\Gamma_j))$. Et donc $E_j^*(t_2+1) \geq E_j^*(t_1+1)$.
  2. Le réservoir n'est pas plein ni à t_1, ni à t_2 . Dans ce cas, dans les deux séquences, le réservoir se remplit de la même quantité égale à P_r et nous avons aussi $E_j^*(t_2+1) = E_j^*(t_1+1) + E_j^*(kH^*(\Gamma_j)) - E_j^*((k-1)H^*(\Gamma_j))$. Et donc $E_j^*(t_2+1) \geq E_j^*(t_1+1)$.
  3. Le réservoir n'est pas plein à t_1 mais plein à t_2 . Soit $E_j^*(t_1) < B_j$ et $E_j^*(t_2) = B_j$. Nous avons donc $E_j^*(t_2+1) = B_j$ car l'énergie récupérée entre t_2 et t_2+1 est gaspillée.

Et nous avons $E_j^*(t_1+1) \leq B_j$. Et donc $E_j^*(t_2+1) \geq E_j^*(t_1+1)$.

Nous appliquons ce raisonnement par récurrence et montrons que $E_j^*((k+1)H^*(\Gamma_j)) \geq E_j^*(kH^*(\Gamma_j))$. Ceci implique que s'il existe un ordonnancement faisable pour τ sur $[0, H^*(\Gamma_j)]$ et si $E_j^*(H^*(\Gamma_j)) \geq E_j^*(0)$, alors le même ordonnancement faisable existe pour sur $[H^*(\Gamma_j), 2H^*(\Gamma_j)]$ tel que $E_j^*(2H^*(\Gamma_j)) \geq E_j^*(H^*(\Gamma_j))$ et donc tel que $E_j^*(2H^*(\Gamma_j)) \geq E_j^*(0)$. Et par récurrence, il existe le même ordonnancement faisable sur $[(k-1)H^*(\Gamma_j), kH^*(\Gamma_j)]$ pour tout k, tel que $E_j^*(kH^*(\Gamma_j)) \geq E_j^*(0)$. Donc, il existe un ordonnancement faisable qui est cyclique de période $H^*(\Gamma_j)$.□

Dans le cas d'un ensemble de tâches asynchrones, nous avons montré (cf. Théoreme 20) que la charge de traitement est la même d'une fenêtre à l'autre (de taille $\Delta^* = 2 \times (2H^*(\Gamma_j) + \Phi)$). Ce résultat va nous permettre d'avancer une condition suffisante garantissant l'ordonnançabilité dans le cas d'un profil énergétique constant avec des batteries initialement pleines :

Théorème 30 *Un ensemble Γ_j de n tâches périodiques asynchrones à échéances contraintes possèdant un facteur d'utilisation énergétique U_{e_j} est ordonnançable sur un processeur π_j considérant un profil énergétique constant si toutes les échéances des jobs rouges sont respectées dans l'intervalle $[0, 2*(max_{i=1}^{n}(O_i)+2H^*(\Gamma_j))]$ sans qu'à aucun moment il n'y ait famine énergétique et qu'à la fin de cet intervalle le niveau d'énergie dans la batterie soit supérieur ou égal au niveau initial.*

Preuve:

Seulement si

L'ensemble de tâches est ordonnançable sur $[(k-1)\Delta^*, k\Delta^*]$, donc tous les jobs rouges réveillés dans cet intervalle ont terminé leur exécution avant échéance sans famine énergétique ($E_j^*(t) \geq 0$). A l'instant $k\Delta^*$, le système, du point de vue de l'activité du processeur se retrouve dans le même état qu'à $(k-1)\Delta^*$. Du point de vue énergétique, le système se retrouve à l'instant $k\Delta^*$ dans une situation plus favorable qu'à $(k-1)\Delta^*$ avec un surplus d'énergie dans le réservoir à l'instant $k\Delta^*$, égal à $E_j^*(k\Delta^*) - E_j^*((k+1)\Delta^*)$. Donc l'ensemble sera aussi ordonnançable sur $[k\Delta^*, (k+1)\Delta^*]$. Montrons maintenant que nous pouvons appliquer aux jobs réveillés sur $[k\Delta^*, (k+1)\Delta^*]$ le même ordonnancement que celui appliqué aux jobs réveillés sur $[(k-1)\Delta^*, k\Delta^*]$ et nous aurons alors nécessairement $E_j^*((k+1)\Delta^*) \geq E_j^*(k\Delta^*)$. Soient deux instants t_1 et t_2 tels que $(k-1)\Delta^* \leq t1 \leq k\Delta^*$ et $t_2 = t_1 + 2\Delta^*$. Supposons que les deux séquences soient identiques respectivement jusqu'à t_1 et t_2 avec $E_j^*(t_2) \geq E_j^*(t_1)$. Montrons qu'elles peuvent être identiques dans l'unité de temps suivante et que nous aurons aussi $E_j^*(t_2+1) \geq E_j^*(t_1+1)$. Entre t_1 et t_1+1, voici les différentes situations possibles :

- Le processeur est actif. Alors ayant davantage d'énergie dans le réservoir qu'à t_1, il peut aussi être actif entre t_2 et t_2+1. Et le niveau d'énergie dans le réservoir a diminué de la même quantité. D'où $E_j^*(t_2+1) = E_j^*(t_1+1) + E_j^*(k\Delta^*) - E_j^*((k-1)\Delta^*)$. Et donc $E_j^*(t_2+1) \geq E_j^*(t_1+1)$.
- Le processeur est inactif. Il peut évidemment aussi être inactif entre t_2 et t_2+1. Trois situations sont toutefois à considérer du point de vue du réservoir :
  1. Le réservoir est plein à t_1 et de l'énergie est gaspillée entre t_1 et t_1+1 d'une quantité égale à Pr . La même quantité sera donc aussi gaspillée entre t_2 et t_2+1 et donc avec la même différence de quantité d'énergie dans le réservoir. Soit $E_j^*(t_2+1) = E_j^*(t_1+1) + E_j^*(k\Delta^*) - E_j^*((k-1)\Delta^*)$. Et donc $E_j^*(t_2+1) \geq E_j^*(t_1+1)$.
  2. Le réservoir n'est pas plein ni à t_1, ni à t_2 . Dans ce cas, dans les deux séquences, le réservoir se remplit de la même quantité égale à P_r et nous avons aussi $E_j^*(t_2+1) = E_j^*(t_1+1) + E_j^*(k\Delta^*) - E_j^*((k-1)\Delta^*)$. Et donc $E_j^*(t_2+1) \geq E_j^*(t_1+1)$.
  3. Le réservoir n'est pas plein à t_1 mais plein à t_2 . Soit $E_j^*(t_1) < B_j$ et $E_j^*(t_2) = B_j$. Nous avons donc $E_j^*(t_2+1) = B_j$ car l'énergie récupérée entre t_2 et t_2+1 est gaspillée. Et nous avons $E_j^*(t_1+1) \leq B_j$. Et donc $E_j^*(t_2+1) \geq E_j^*(t_1+1)$.

Nous appliquons ce raisonnement par récurrence et montrons que $E_j^*((k+1)\Delta^*) \geq E_j^*(k\Delta^*)$. Ceci implique que s'il existe un ordonnancement faisable pour τ sur $[0, \Delta^*]$ et si $E_j^*(\Delta^*) \geq E_j^*(0)$, alors le même ordonnancement faisable existe pour sur $[\Delta^*, 2\Delta^*]$ tel que $E_j^*(2\Delta^*) \geq E_j^*(\Delta^*)$ et donc tel que $E_j^*(2\Delta^*) \geq E_j^*(0)$. Et par récurrence, il existe le même ordonnancement faisable sur $[(k-1)\Delta^*, k\Delta^*]$ pour tout k, tel que $E_j^*(k\Delta^*) \geq E_j^*(0)$. Donc, il existe un ordonnancement faisable qui est cyclique de période Δ^*.□

3 Extension de AA_{QoS} à des systèmes autonomes énergétiquement

Dans cette partie, nous étendons l'heuristique de partitionnement AA_{QoS} à des tâches temps réel fermes pour prendre en compte cette fois-ci les contraintes énergétiques.

3.1 Description algorithmique

La Figure 5.2 montre le pseudo-code d'une heuristique de partitionnement AA_{QoS} de tâches fermes adapté aux systèmes autonomes énergétiquement que l'on dénote $AA_{QoS} - AS$ ($AA_{QoS} - AS$ peut être $FF_{QoS} - AS$, $BF_{QoS} - AS$, $WF_{QoS} - AS$ ou encore $NF_{QoS} - AS$). Pour chaque processeur π_j, Γ_j dénote les tâches parmi τ_1,..., τ_{i-1} qui ont déja été assignées au processeur π_j. Initialement, $\forall j = 1..m, \Gamma_j = \varnothing$.

Algorithme 4 Algorithme $AA_{QoS} - AS$

Entrées : $m, \tau = \{\tau_1, ..., \tau_n\}$

Sorties : SUCCES et $\{\Gamma_1, ..., \Gamma_m\}$ si τ est ordonnançable, ECHEC sinon.

Fonction : $FF_{QoS} - AS$

```
début
pour i = 1 → |τ| faire
    /* Tâche assignée selon l'heuristique AA, ici FF est utilisé */
    Ordo ← faux ;
    pour j = 1 → m faire
        si jobs rouges de τ_i sont temporellement-ordonnançable et énergétiquement-ordonnançable sur π_j
alors
            /* τ_i est assignée au processeur π_j ; */
            Γ_j = Γ_j ∪ τ_i ;
            Ordo ← vrai ;
            break ;
        fin si
    fin pour
    si not Ordo alors
        retourner ECHEC ;
    fin si
fin pour
retourner SUCCES ;
fin
```

FIGURE 5.2 – Pseudo-code de $AA_{QoS} - AS$

Théorème 31 *Si l'algorithme $AA_{QoS} - AS$ retourne SUCCES pour un ensemble de tâches τ à échéances contraintes, alors le partitionnement obtenu en utilisant l'heuristique AA est ordonnançable considérant un modèle purement rouge.*

Preuve:
On observe que l'algorithme retourne SUCCES si et seulement s'il a assigné avec succès chaque tâche de τ aux différents processeurs. Avant l'assignation de la tâche τ_i, l'ensemble des jobs rouges des tâches de chaque processeur est trivialement ordonnançable. Il résulte du Théorème 29 (ou 30 pour des tâches asynchrones) que tous les ensembles de tâches des processeurs restent ordonnançables après chaque attribution de tâche. Par conséquent, tous les ensembles de tâches des processeurs sont ordonnançables une fois que toutes les tâches de τ ont été assignées.□

4 Extension de $KTS - AA_{QoS}$ à des systèmes autonomes énergétiquement

4.1 Principe

Nous étendons l'approche $KTS-AA_{QoS}$ aux systèmes autonomes énergétiquement, désormais dénotée $KTS - AA_{QoS} - AS$ pour permettre la répartition d'ensemble de tâches avec respect des contraintes temporelles tout comme des contraintes d'énergie et avec une étape de fractionnement en cas de rejet d'une tâche.

Le principe de fonctionnement de l'algorithme $KTS-AA_{QoS}-AS$ est identique à celui décrit dans le chapitre 3. Seulement cette fois-ci, les tests d'ordonnnançabilité incluent le respect des contraintes temporelles et énergétiques liées à l'exécution des jobs rouges uniquement. Les tâches sont d'abord assignées aux différents processeurs selon une certaine heuristique. Dès lors que l'allocation d'une tâche sur un processeur n'est plus possible (un rejet dû à (i) une violation d'échéance d'un job rouge, (ii) à la présence d'une famine énergétique et/ou (iii) à un niveau d'énergie dans la batterie à la fin de la fenêtre de test en deça du niveau initial), la tâche rejetée va subir un fractionnement selon l'algorithme $KTS - AA_{QoS} - AS$, comme décrit dans les Figures 3.2 et 3.3 (page 61).

4.2 Description algorithmique

La Figure 5.3 montrent le pseudo-code de l'algorithme $KTS - FF_{QoS} - AS$. k représente le niveau de *task splitting* courant atteint et $maxK$ est la profondeur limite de *task splitting*. Sur une plateforme multiprocesseur, $\tau(\pi)$ désigne l'ensemble de tâches parmi τ_1,..., τ_{i-1} ayant été assignées avec succès à l'un des processeurs de l'ensemble π. Initialement, $\tau(\pi) = \varnothing$.

Au début, l'algorithme fonctionne selon une heuristique de partitionnement donnée. Si aucun processeur n'est capable de recevoir une certaine tâche, nous ne pouvons rien conclure quant à l'ordonnançabilité de l'ensemble de tâches τ sur la plateforme multiprocesseur. La tâche τ_i est alors fractionnée en un sous-ensemble τ_i^{split} composé de a tâches (voir Chapitre 3) qui sont assignées à un sous-ensemble de processeurs de la plateforme à m processeurs. Le Lemme 5.1 affirme qu'en attribuant une tâche fractionnée de τ_i^{split} à un processeur π_j, l'ordonnançabilité des tâches confiées précédemment à l'ensemble des processeurs reste intacte.

Lemme 5.1 *Si l'ensemble de tâches précédemment assignées aux processeurs est ordonnançable (respect des contraintes temporelles et énergétiques sur chaque processeur) et que par la suite l'algorithme $KTS - AA_{QoS} - AS$ assigne une tâche τ_i^0 (respectivement τ_i^1) au processeur π_j (d'après le Théoreme 30), alors l'ensemble de tâches assignées à chaque processeur (y compris processeur π_j) reste ordonnançable.*

Preuve: On observe que l'ordonnançabilité des processeurs autres que le processeur π_j n'est pas affectée par l'allocation de la tâche τ_i^0 (respectivement τ_i^1) au processeur π_j. Aussi, si l'ensemble

de tâches précédemment assigné à π_j était ordonnançable sur π_j avant l'allocation de τ_i^0 (respectivement τ_i^1) du fait que tous leurs jobs rouges s'exécutent avant leurs échéances sans qu'il n'y ait à aucun moment une famine énergétique et que la condition du Théoreme 30 est satisfaite, alors l'ensemble de tâches sur π_j reste ordonnançable après l'ajout de τ_i^0 (respectivement τ_i^1).□

Le fonctionnement de $KTS\text{-}AA_{QoS}\text{-}AS$ détermine, par des applications répétées du Lemme 5.1 au niveau de chaque tâche à assigner, l'ordonnançabilité de l'ensemble de tâches.

Algorithme 5 Algorithme de partitionnement $KTS - FF_{QoS} - AS$

Entrées : $m, k, maxK, \tau = \{\tau_1, ..., \tau_n\}$

Sorties : SUCCES et $\tau(\pi)$ si τ est ordonnançable, ECHEC sinon.

Fonction : $KTS - FF_{QoS} - AS$

```
début
pour i = 1 → |τ| faire
    Ordo ← faux ;
    /*Attribution de la tâche τ_i selon FFD_QoS */
    pour j = 1 → m faire
        si jobs rouges de τ_i sont temporellement-ordonnançable et énergétiquement-ordonnançable sur π_j
alors
            /* τ_i est attribuée à π_j */
            τ(π) = τ(π) ∪ τ_i ;
            Ordo ← vrai ;
            break ;
        fin si
    fin pour
    si not Ordo alors
        /* Essai de fractionnement de la tâche */
        si k < maxK alors
            /* La profondeur limite de fractionnement maxK n'est pas atteinte*/
            /* τ_i(O_i, C_i, T_i, D_i, s_i) est fractionnée en un ensemble composé de a sous-tâches */
            si s_i = ∞ alors
                τ_i^split = {τ_i^0(O_i + x × T_i, C_i, 2 × T_i, D_i, ∞), x = 0..1} ; // a = 2
            sinon
                si s_i = 2 alors
                    τ_i^split = {τ_i^0(O_i + x × s_i T_i, C_i, 2 × s_i T_i, D_i, ∞), x = 0..1} ; // a = 2
                sinon
                    τ_i^split = {τ_i^x(O_i + x × T_i, C_i, s_i × T_i, D_i, ∞), x = 0..(s_i − 2)} // a = s_i − 1
                fin si
            fin si
            si KTS − FF_QoS − AS(m, k + 1, maxK, τ_i^split) == SUCCES alors
                Ordo ← vrai ;
            fin si
        sinon
            /* La profondeur limite de fractionnement maxK a été atteinte*/
            Ordo ← faux ;
        fin si
    fin si
    si not Ordo alors
        retourner ECHEC ;
    fin si
fin pour
retourner SUCCES ;
fin
```

FIGURE 5.3 – Pseudo-code de $KTS - FF_{QoS} - AS$

Théorème 32 *Considérant un ensemble de tâches périodiques τ, si l'algorithme de partitionnement $KTS{-}AA_{QoS}{-}AS$ retourne SUCCES, alors le partitionnement établi est ordonnançable.*

Preuve: On observe que l'algorithme retourne SUCCES si et seulement s'il a assigné avec succès chaque tâche de τ aux différents processeurs. Avant l'assignation de la tâche τ_i, l'ensemble de tâches de chaque processeur est trivialement ordonnançable. Il résulte du Lemme 5.1 que tous les ensembles de tâches des processeurs restent ordonnançables après chaque attribution de tâche. Par conséquent, tous les ensembles de tâches des processeurs sont ordonnançables une fois que toutes les tâches de τ ont été assignées.□

5 Validation par simulation

L'objectif de ce paragraphe est de rapporter les différentes simulations effectuées dans le but d'analyser le comportement des heuristiques décrites dans des cas de surcharge énergétique et/ou de traitement, tout en analysant l'effet du paramètre de pertes s_i.

5.1 Environnement de simulation

L'environnement de simulation consiste en m ordonnanceurs associés aux différents processeurs constituant la plateforme. Le programme de simulation a été implémenté en langage Perl. L'architecture fonctionnelle du simulateur est représentée sur la Figure 5.4.

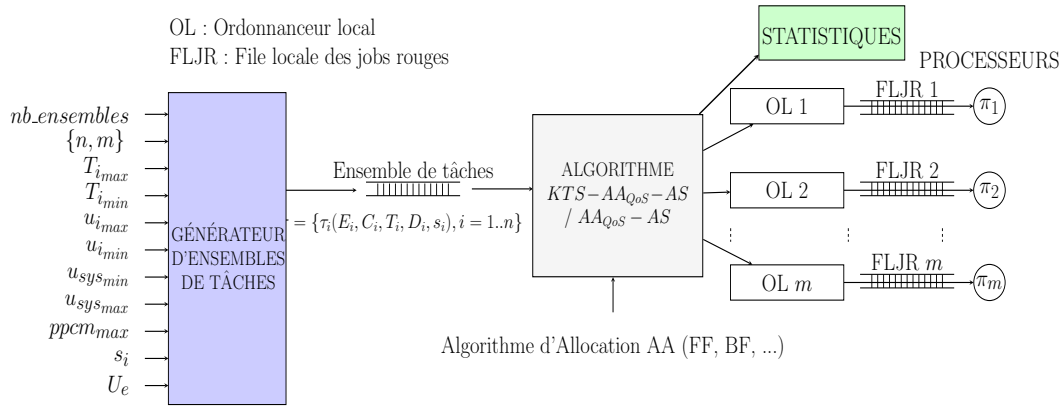


FIGURE 5.4 – Architecture fonctionnelle du simulateur

Le générateur d'ensembles de tâches périodiques a été conçu de manière à ce que les facteurs d'utilisation u_i des tâches de l'ensemble généré soient répartis de manière uniforme dans $[u_{i_{min}}, u_{i_{max}}] = [0.1, 1]$. Il accepte en entrée plusieurs paramètres : le nombre de processeurs m de la plateforme, le nombre de tâches n à partitionner, le nombre d'ensembles de tâches généré *nb_ensembles* pour chaque configuration, le ppcm maximum $ppcm_{max}$ des périodes des tâches choisies dans l'intervalle $[T_{i_{min}}, T_{i_{max}}] = [10, 200]$ et les facteurs de pertes *si*. La charge périodique totale du système (composé de m processeurs) u_{sys} varie dans $[u_{sys_{min}}, u_{sys_{max}}] = [0.3, 1.4]$.

En sortie, on obtient pour chaque valeur de u_{sys}, *nb_ensembles* ensembles de tâches périodiques synchrones $\tau = \{\tau_i(E_i, C_i, T_i, D_i, s_i), i = 1..n\}$. Les durées d'exécution des tâches sont générées à partir des facteurs d'utilisation et des périodes affectés aux tâches, $C_i = u_i \times T_i$. Les consommations énergétiques pire-cas E_i des tâches sont choisies aléatoirement de manière à ce que leur somme soit égale au facteur d'utilisation énergétique totale U_e fourni en entrée.

Dans notre méthode de génération de tâches, nous considérons : (i) des tâches à échéances contraintes (D_i est choisie aléatoirement dans $[C_i, T_i]$) et (ii) des tâches à échéances sur requêtes ($D_i = T_i$).
Après la génération des différents ensembles de tâches, le simulateur tente de partitionner, pour chaque valeur de u_{sys}, les *nb_ensembles* ensembles de tâches et détermine la proportion d'ensembles de tâches jugé ordonnançable appelé le *taux de réussite* (*success ratio* en anglais). Le taux de réussite est le critère de performance qui permettra de comparer les différentes stratégies entre elles.

Dans toutes nos simulations, nous générons 100 ensembles de 24 tâches possédant différents niveaux de criticité d'énergie R_e ($U_e = m.P_r.R_e$). Le nombre de processeurs est fixé à 16 processeurs.

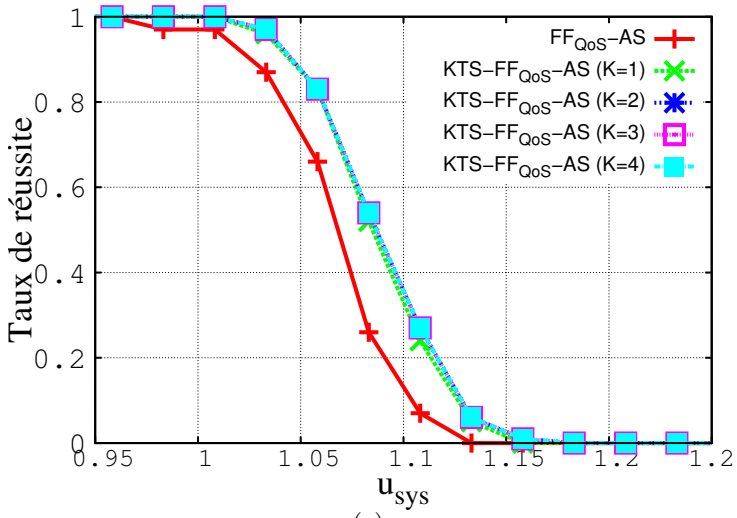
5.2 Évaluation des algorithmes développés

Nous cherchons à évaluer les performances $KTS - AA_{QoS} - AS$ et $AA_{QoS} - AS$ pour des systèmes possédant différents niveaux de criticité d'énergie $R_e = \{0.3, 0.75, 1.2\}$ selon différentes valeurs de s_i. Nous présentons ci-dessous les résultats de simulations considérant deux cas : (a) modèle de tâches à échéances sur requêtes $D = T$, (b) modèle de tâches à échéances contraintes $D \leq T$. L'ordre de performances obtenues étant le même quel que soit l'algorithme d'allocation choisi, nous présenterons ceux où $AA = FF$.

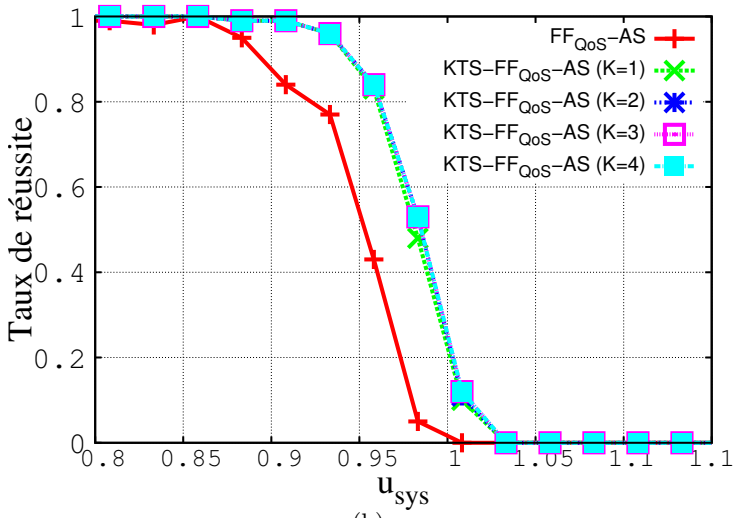
5.2.1 Effet du paramètre de pertes s_i

Les Figures 5.5, 5.6, et 5.7 illustrent le taux de réussite de $FF_{QoS} - AS$ et $KTS - FF_{QoS} - AS$ selon s_i pour des ensembles de tâches à échéances sur requêtes sous respectivement trois niveaux de criticité d'énergie différents. Notons que la Figure 5.7 représente un cas de surcharge énergétique ($R_e = 1.2 > 1$).

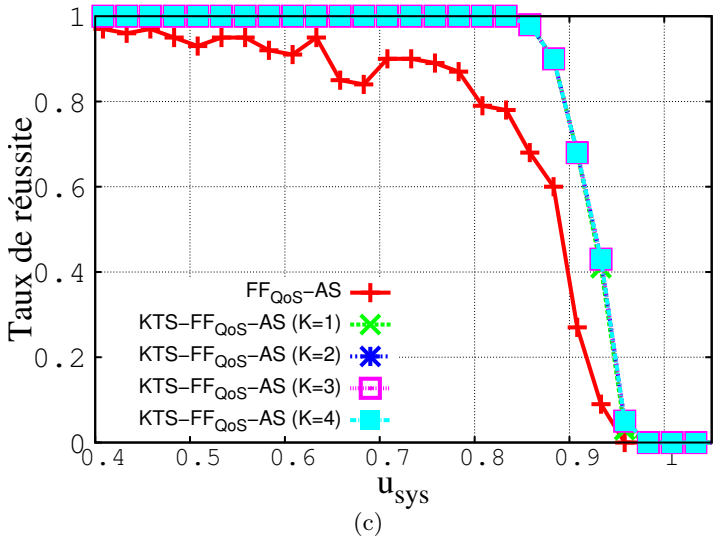
Les Figures 5.8, 5.9, et 5.10 illustrent le taux de réussite de $FF_{QoS} - AS$ et $KTS - FF_{QoS} - AS$ selon s_i pour des ensembles de tâches à échéances contraintes sous respectivement trois niveaux de criticité d'énergie différents.



(a)

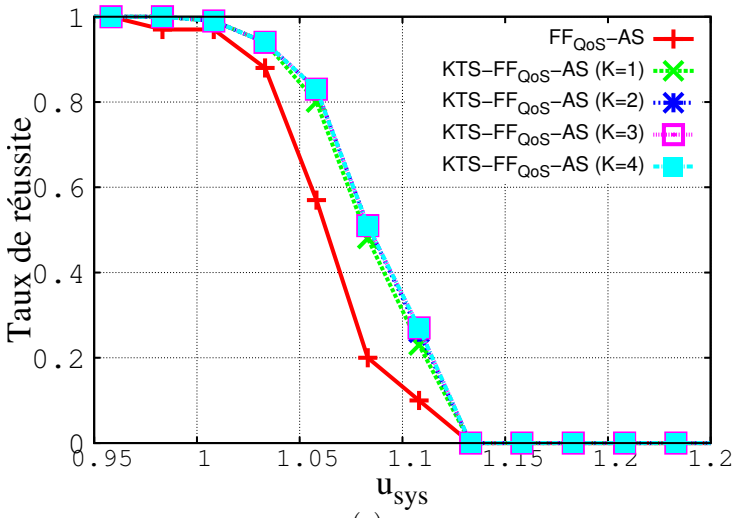


(b)

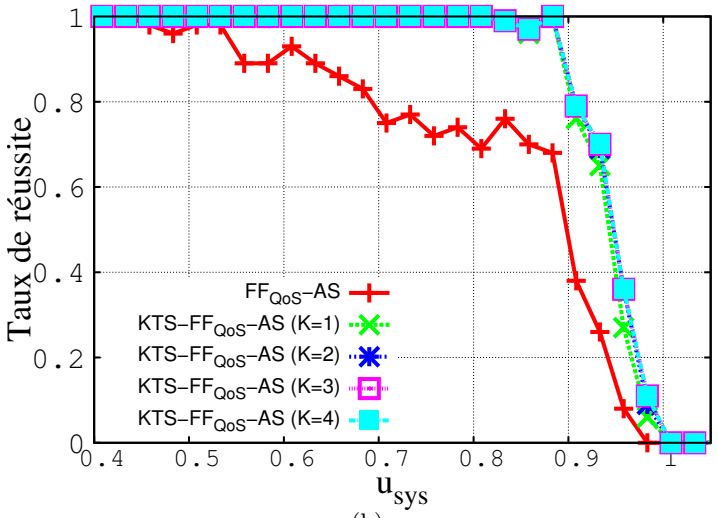


(c)

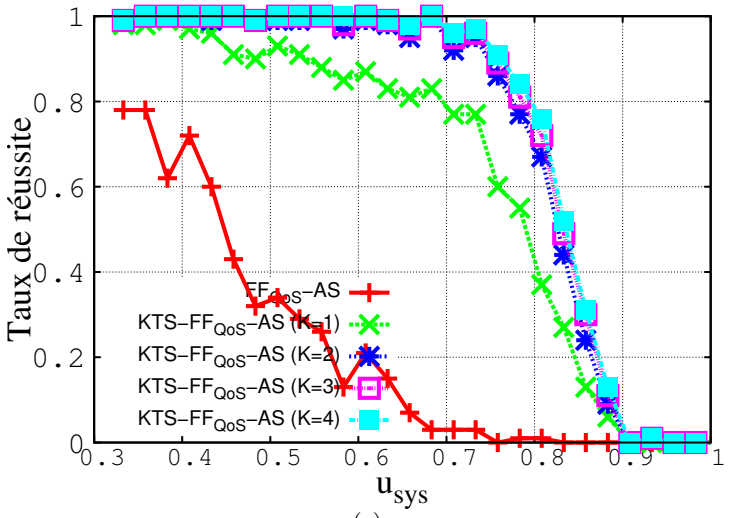
FIGURE 5.5 – Performance de $FF_{QoS} - AS$ et $KTS - FF_{QoS} - AS$ pour des systèmes faiblement contraints énergétiquement ($R_e = 0.3$) avec $D = T$ et : (a) $s_i = 2$ (50% de pertes autorisées), (b) $s_i = 4$ (25% de pertes autorisées), (c) $s_i = \infty$ (0% de pertes autorisées)



(a)



(b)



(c)

FIGURE 5.6 – Performance de FF_{QoS}−AS et KTS−FF_{QoS}−AS pour des systèmes moyennement contraints énergétiquement (R_e = 0.75) avec $D = T$ et : (a) s_i = 2 (50% de pertes autorisées), (b) s_i = 4 (25% de pertes autorisées), (c) $s_i = \infty$ (0% de pertes autorisées)

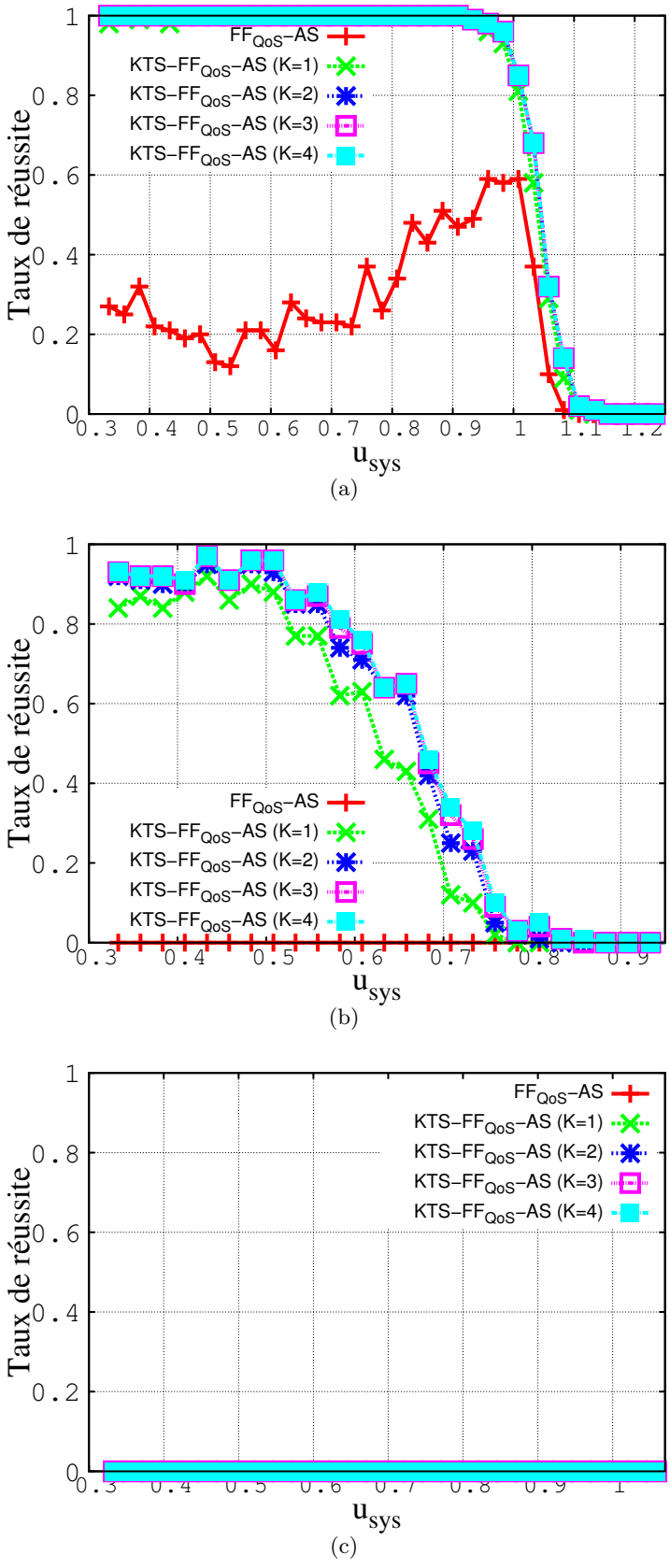
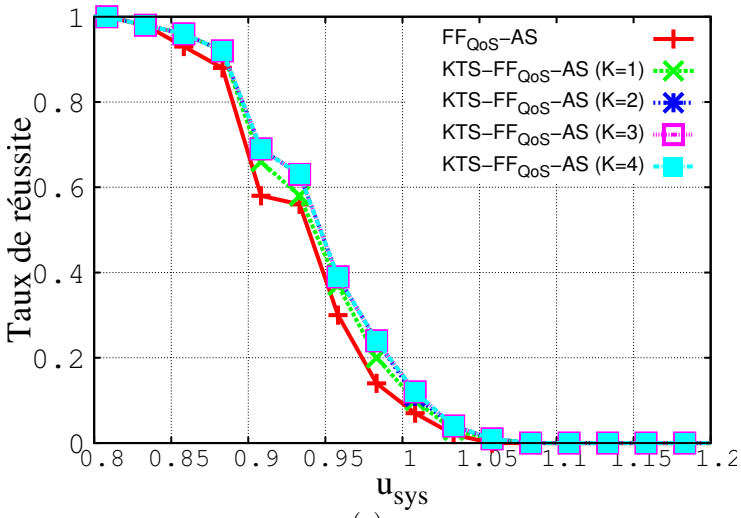
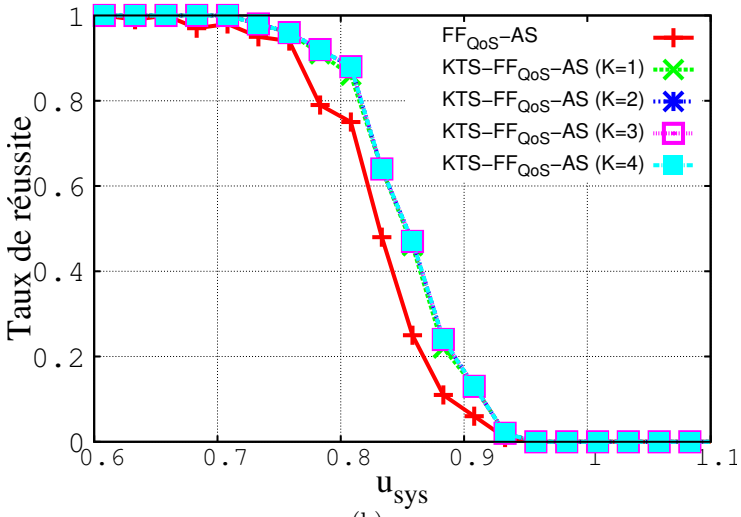


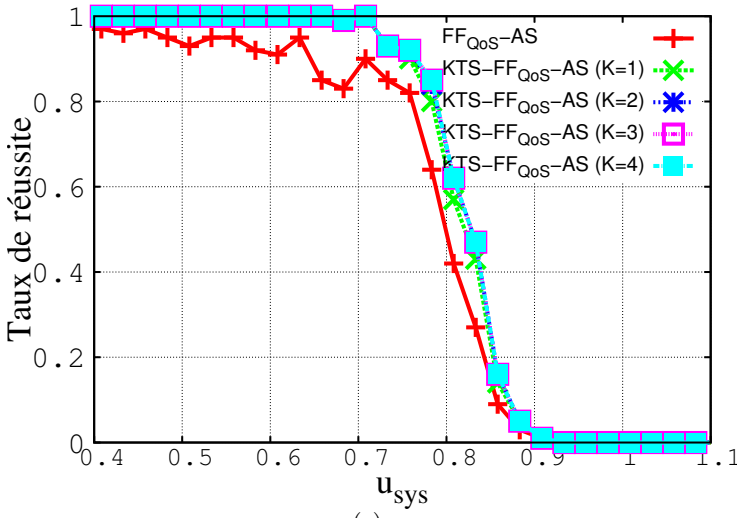
FIGURE 5.7 – Performance de $FF_{QoS}-AS$ et $KTS-FF_{QoS}-AS$ pour des systèmes en surcharge d'énergie (R_e = 1.2) avec $D = T$ et : (a) s_i = 2 (50% de pertes autorisées), (b) s_i = 4 (25% de pertes autorisées), (c) $s_i = \infty$ (0% de pertes autorisées)



(a)

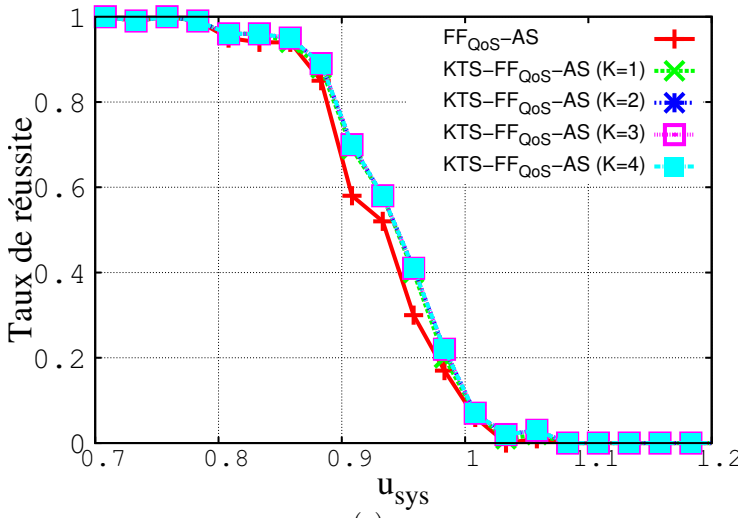


(b)

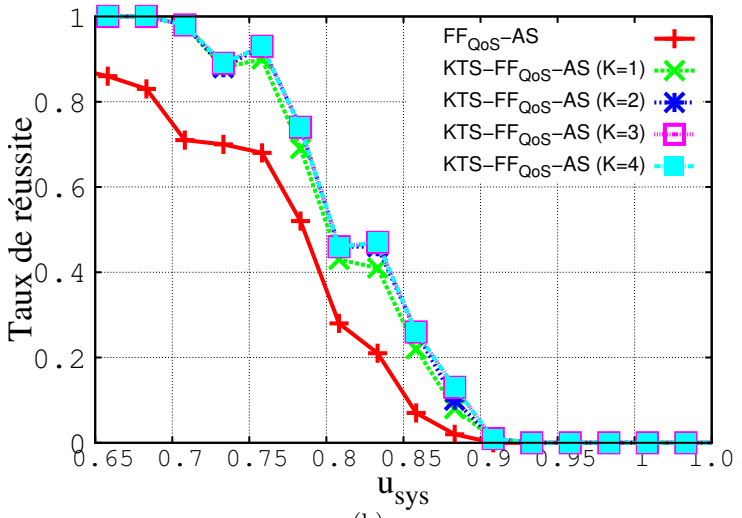


(c)

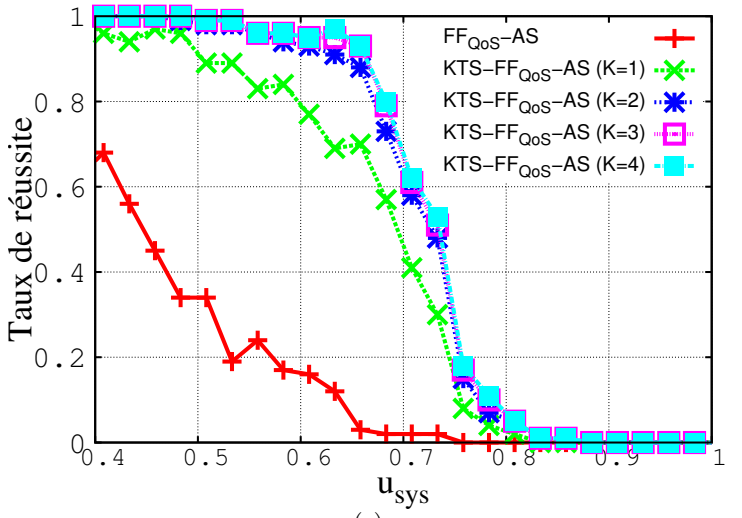
FIGURE 5.8 – Performance de $FF_{QoS} - AS$ et $KTS - FF_{QoS} - AS$ pour des systèmes faiblement contraints énergétiquement ($R_e = 0.3$) avec $D \leq T$ et : (a) $s_i = 2$ (50% de pertes autorisées), (b) $s_i = 4$ (25% de pertes autorisées), (c) $s_i = \infty$ (0% de pertes autorisées)



(a)

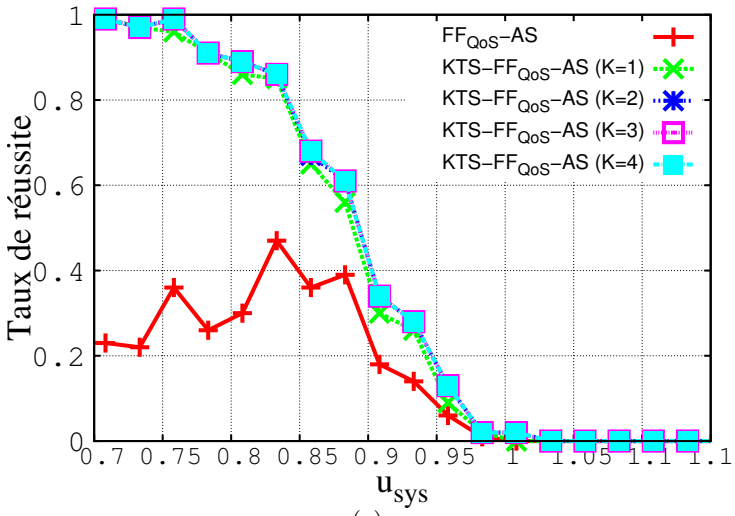


(b)

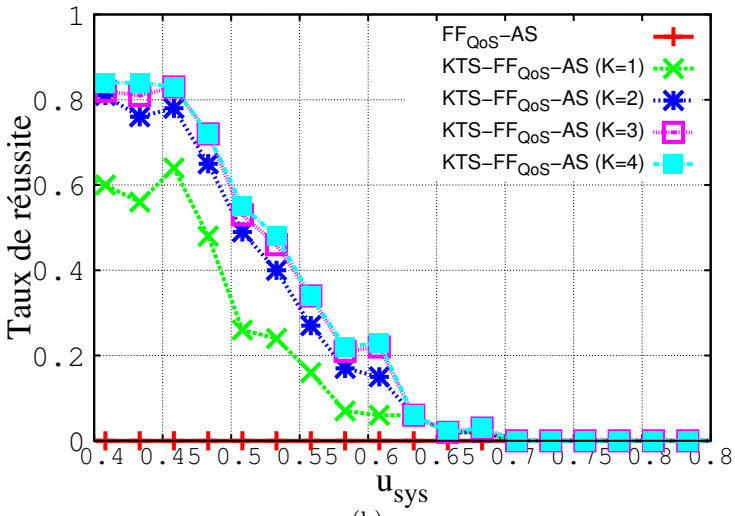


(c)

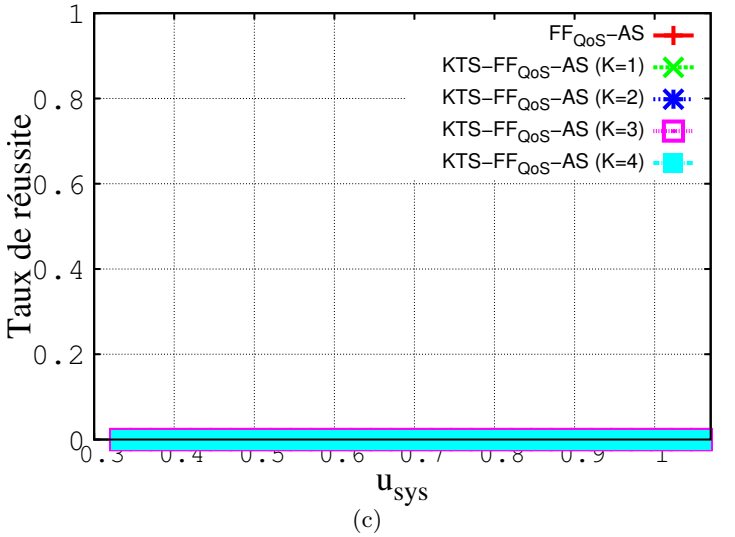
FIGURE 5.9 – Performance de $FF_{QoS}-AS$ et $KTS-FF_{QoS}-AS$ pour des systèmes moyennement contraints énergétiquement (R_e = 0.75) avec $D \leq T$ et : (a) s_i = 2 (50% de pertes autorisées), (b) s_i = 4 (25% de pertes autorisées), (c) $s_i = \infty$ (0% de pertes autorisées)



(a)



(b)



(c)

FIGURE 5.10 – Performance de $FF_{QoS}-AS$ et $KTS-FF_{QoS}-AS$ pour des systèmes en surcharge d'énergie (R_e = 1.2) avec $D \leq T$ et : (a) s_i = 2 (50% de pertes autorisées), (b) s_i = 4 (25% de pertes autorisées), (c) $s_i = \infty$ (0% de pertes autorisées)

Les résultats des Figures 5.5, 5.6, et 5.7 tout comme celles des Figures 5.8, 5.9, et 5.10 montrent que $KTS - AA_{QoS} - AS$ offre de meilleures performances et ce, quels que soient l'algorithme d'allocation, le niveau de criticité du système, le modèle de tâches et le paramètre de pertes considérés. Il permet d'augmenter progressivement le taux de réussite à mesure que l'on augmente la profondeur limite K jusqu'à une certaine limite (cf. page 64). De plus, nous notons que les écarts de performances les plus grands entre $KTS - FF_{QoS} - AS$ et $FF_{QoS} - AS$ sont obtenus pour des systèmes moyennement contraints énergétiquement, voire en surcharge énergétique. Nous remarquons également que l'écart de performances entre $KTS - FF_{QoS} - AS$ et $FF_{QoS} - AS$ quel que soit R_e augmente à mesure que s_i augmente. En effet, la flexibilité apportée par un petit s_i suffit parfois à rendre un ensemble de tâches ordonnançable. Il n'y a alors pas besoin de fractionner les tâches. Cependant, nous remarquons que pour un niveau maximum de pertes autorisées ($s_i = 2$), et lorsque le système est en surcharge énergétique, FF_{QoS} arrive à partitionner une faible proportion d'ensembles de tâches comparé à $KTS - AA_{QoS} - AS$ en particulier pour des systèmes faiblement chargé. Nous observons par exemple pour $s_i = 2$, $R_e = 1.2$ et $u_{sys} = 0.53$, lorsque $D = T$, $FF_{QoS} - AS$ arrive à ordonnancer 13% des ensembles de tâches générés tandis que 100% des ensembles de tâches sont ordonnançables en utilisant $KTS - FF_{QoS} - AS$ avec $K = 4$. En effet, lorsque l'utilisation du système est faible comparé au niveau de criticité R_e, comme les facteurs d'utilisation sont plus petits et que les facteurs d'utilisation énergétique, eux, sont importants, une tâche, qui s'exécute, consomme plus en une unité de temps faisant ainsi chuter brutalement le niveau d'énergie dans la batterie. $KTS - AA_{QoS} - AS$ est donc plus performant par le fait qu'il crée de nouvelles tâches avec des facteurs d'utilisation énergétique réduis.

6 Conclusion

Dans ce chapitre, nous nous sommes intéressés au problème du partitionnement de tâches fermes sur des plateformes multiprocesseur autonomes énergétiquement. Nous avons abordé le problème de la répartition d'un ensemble de tâches, tolérant un certain ratio de pertes (selon le modèle Skip-Over), sur un système multiprocesseur alimenté par une source d'énergie renouvelable. Une analyse d'ordonnançabilité a été présentée pour prendre en compte à la fois les contraintes temporelles et d'énergie pour des tâches périodiques considérant un modèle profondément rouge. Nous avons ensuite étendu cette analyse à un modèle de tâches périodiques fermes asynchrones.

L'évaluation sur une large gamme d'ensembles de tâches indique que $KTS - AA_{QoS} - AS$ offre des performances meilleures que celles de $AA_{QoS} - AS$. En effet, comparativement aux heuristiques adaptées $AA_{QoS} - AS$, à mesure que l'on augmente la profondeur limite K, la performance de $KTS - AA_{QoS} - AS$ s'améliore et l'écart devient plus significatif. Nous avons également observé que $KTS - AA_{QoS} - AS$ est plus performant lorsque le taux de pertes autorisées est faible et/ou que le niveau de criticité énergétique est important.

Synthèse générale

Le Tableau 5.1 fournit un guide de choix des heuristiques de partitionnement étudiées tout au long de la thèse en fonction des contraintes liées au système considéré :

Heuristique de partitionnement	Classique	Avec QoS	Avec Énergie
AA	√	-	-
$AA - AS$	-	-	√
AA_{QoS}	-	√	-
$AA_{QoS} - AS$	-	√	√
$KTS - AA$	√	-	-
$KTS - AA - AS$	-	-	√
$KTS - AA_{QoS}$	-	√	-
$KTS - AA_{QoS} - AS$	-	√	√

TABLE 5.1 – Choix des algorithmes en fonction des contraintes du système considéré

Nous citons par exemple, le cas d'un système temps réel sous contraintes d'énergie et de QoS, le concepteur peut utiliser $AA_{QoS} - AS$ ou $KTS - AA_{QoS} - AS$ comme heuristique de partitionnement. Cependant de manière à obtenir les meilleures performances, la décision est prise en fonction du ratio de tâches lourdes, du nombre de processeurs, de la QoS du système considéré. Le Tableau 5.2 fournit, en fonction des paramètres du système considéré, un guide de choix des heuristiques de partitionnement étudiées tout au long de la thèse :

<table>
<tr><td colspan="4">ratio de tâches lourdes ≥ 50%</td><td rowspan="3">$m < 4$</td><td rowspan="3">ratio de tâches lourdes < 50%</td></tr>
<tr><td colspan="4">$m \geq 4$</td></tr>
<tr><td colspan="2">Sans Énergie</td><td colspan="2">Avec Énergie</td></tr>
<tr><td>Avec QoS</td><td>Sans QoS</td><td>Avec QoS</td><td>Sans QoS</td><td></td><td></td></tr>
<tr><td colspan="5">KTS adapté</td><td>AA adapté</td></tr>
<tr><td>$KTS - AA_{QoS}$</td><td>$KTS - AA$</td><td>$KTS - AA_{QoS} - AS$</td><td>$KTS - AA - AS$</td><td colspan="2">– AA / AA_{QoS} (sans/avec QoS respectivement)
– $AA - AS$ (avec Énergie)
– $AA_{QoS} - AS$ (avec Énergie et QoS))</td></tr>
</table>

TABLE 5.2 – Choix des algorithmes en fonction des paramètres du système considéré

Supposons maintenant que le concepteur dispose d'un système temps réel à 8 processeurs sous contraintes énergétiques destiné à exécuter des ensembles composés à 70% de tâches lourdes et autorisant 25% de pertes, la méthode $KTS - AA_{QoS}$ est choisie comme étant la plus efficace. Cependant, pour ce même système, si le ratio de tâches lourdes est inférieur à 50%, les performances de KTS adapté sont légèrement supérieures ou égales à celles de AA adapté mais possèdent un temps de calcul plus important et il vaudra donc mieux utiliser AA adapté. Ceci est aussi le cas pour une plateforme composée de moins de 4 processeurs.

Conclusion générale

Ce rapport a proposé des solutions d'ordonnancement efficaces et simples d'implémentation pour des systèmes temps réel multiprocesseur homogènes autonomes énergétiquement permettant d'assurer une dégradation contrôlée de la QoS, en particulier dans des phases de famine énergétique et/ou de surcharge de traitement. Après avoir mené une analyse des concepts relatifs à l'ordonnancement dans les systèmes temps réel multiprocesseur, nous nous sommes intéressés d'une part aux modèles de résolution dédiés à l'ordonnancement de systèmes surchargés et d'autre part aux différentes stratégies liées à l'ordonnancement sous contraintes d'énergie, nous avons dégagé des idées utiles dans les développements réalisés. Dans un premier temps, nous avons proposé une nouvelle stratégie de partitionnement d'un ensemble de tâches temps réel périodiques *à contraintes strictes* basée sur le fractionnement de tâches (*task splitting*). Cette approche dénommée KTS consiste à créer de nouvelles tâches indépendantes plus légères à partir d'une tâche n'ayant pu être assignée. Celle-ci est conceptuellement simple, facile à mettre en œuvre dans les systèmes existants, et efficace de par le fait qu'elle offre des performances significativement meilleures que celles offertes par les heuristiques de bases. L'approche proposée présente l'avantage de n'induire aucun coût de migrations et ce, quels que soient le type de la tâche ($D = T$ ou $D \leq T$, lourde/légère) et le nombre de processeurs du système. Elle offre des performances intermédiaires entre l'approche semi-partitionnée existante EDF split (DD) et l'heuristique de partitionnement classique FFDD. Toutefois, KTS aura tendance à être d'autant plus efficace que le nombre de processeurs est grand et que les ensembles de tâches considérés sont majoritairement composés de tâches lourdes.

L'objectif poursuivi ensuite consistait à étendre, à un modèle de tâches fermes, les heuristiques de partitionnement classiques et l'approche KTS proposée afin d'exploiter la flexibité apportée par le modèle de tâches fermes pour améliorer le taux d'ordonnançabilité. Au travers de ce travail, nous avons montré comment l'analyse étendue d'ordonnançabilité développée dans le contexte de tâches périodiques asynchrones fermes peut être utilisée conjointement à un algorithme de fractionnement de tâches. L'évaluation sur une large gamme d'ensemble de tâches nous a permis de confirmer la supériorité de l'algorithme $KTS - AA_{QoS}$ sur les heuristiques de partitionnement classiques adaptées AA_{QoS}.

Nous nous sommes ensuite intéressés au problème relatif au partitionnement d'un ensemble de tâches temps réel périodiques *à contraintes strictes* sur une plateforme multiprocesseur homogène autonome du point de vue énergétique. Le but a été de répartir les tâches de manière à ce que toutes les contraintes temporelles soient satisfaites sur chaque processeur sans qu'il n'y ait à aucun moment une famine énergétique empêchant l'exécution des jobs. Ce travail a été mené en considérant la politique d'ordonnancement Earliest Deadline First (EDF). Nous avons donc présenté une extension des heuristiques de partitionnement classiques et étendu l'approche KTS pour le système considéré. Au travers des simulations, nous avons exploré la façon dont à la fois les critères de tri des tâches temps réel et le niveau de criticité énergétique peuvent favoriser la performance des différentes heuristiques adaptées. De plus, nous avons estimé une taille de batterie minimale permettant d'obtenir les meilleures performances quels que soient le niveau de contrainte énergétique du système et/ou l'heuristique considérée.

Enfin, nous avons ensuite considéré la prise en compte conjointe de contraintes temporelles et énergétiques pour un modèle de tâches temps réel périodiques fermes *sous contraintes de QoS*.

L'enjeu consistait alors à répartir les tâches en respectant à la fois leurs contraintes temporelles et leurs contraintes énergétiques, tout en assurant l'exécution des jobs obligatoires permettant de garantir la qualité de service minimale requise pour le système.

Les travaux de recherche futurs reposent sur une gestion dynamique de l'exécution des jobs optionnels de manière à assurer une meilleure QoS. En fait, jusqu'ici la répartition des tâches fermes s'est faite de manière à assurer uniquement l'exécution des jobs impératifs (QoS minimale assurée). Il conviendrait alors d'envisager d'ordonnancer les jobs optionnels d'une tâche τ_i en les considérant par exemple comme des tâches apériodiques autorisées à migrer entre les processeurs. L'exécution des jobs rouges ne devrait bien sûr pas être compromise (pas de famine énergétique ni de violations d'échéance).

Publications

Conférences internationales avec comité de lecture

Abdallah, N., Queudet, A., Chetto, M. and Chehade, R.-H. *Partitioned EDF Scheduling in Multicore systems with Quality of Service constraints.* Proceedings of IEEE International Conference on Electronics, Circuits, and Systems (ICECS), Beirut, 2011.

Abdallah, N., Queudet, A. and Chetto, M. *Task Partitioning Strategies for Multicore Real-Time Energy Harvesting Systems.* Proceedings of the IEEE Computer Society Symposium dealing with the rapidly expanding field of object/component/service-oriented real-time distributed computing Technology (ISORC), Nevada, 2014.

Bibliographie

[ABB08] B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems multiprocessors. *In Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 385–394, 2008.

[ABD05] J. Anderson, V. Bud, and U. C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. *In Proceedings of the 17th IEEE Euromicro Conference on Real-Time Systems*, pages 199–208, 2005.

[ABJ01] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. *In Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 193–202, 2001.

[ABR+93] N.C. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings. Applying new scheduling theory to static priority preemptive scheduling. *Software Engineering Journal, 8(5)*, pages 284–292, 1993.

[ABRW91] N.-C. Audsley, A. Burns, M. Richardson, and A.-J. Wellins. Hard real-time scheduling : The deadline monotonic approach. *In Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software, Atlanta*, pages 182–190, 1991.

[AJ00] B. Andersson and J. Jonsson. Fixed-priority preemptive multiprocessor scheduling : to partition or not to partition. *In Proceedings of the International Conference on Real-Time Computing Systems and Applications, Cheju Island, Korea*, 2000.

[AM01] A. Allavena and D. Mosse. Scheduling of frame-based embedded systems with rechargeable batteries. *In Proceedings of the Workshop on Power Management for Real-time and Embedded systems*, 2001.

[And00] J. H. Anderson. Pfair scheduling : Beyond periodic task systems. *In Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications*, pages 297–306, 2000.

[AT06] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. *In Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2006.

[AY03] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. *In Proceedings of 17th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 113–121, 2003.

[BA09] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. *In Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 447–456, 2009.

[BB97] G. Bernat and A. Burns. Combining (n,m)-hard deadlines and dual-priority scheduling. *In Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, pages 46–57, 1997.

[BB05] E. Bini and G.C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems Journal, 30(1-2)*, pages 129–154, 2005.

[BBB01] E. Bini, G.-C. Buttazzo, and G.-M. Buttazzo. A hyperbolic bound for the rate monotonic algorithm. *In Proceedings of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands*, pages 59–66, 2001.

[BBL01] G. Bernat, A. Burns, and A. Llamosí. Weakly hard real-time systems. *In IEEE Transactions on Computers, 50(4)*, pages 308–321, 2001.

[BCPV96] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress : A notion of fairness in resource allocation. *Algorithmica, 15*, pages 600–625, 1996.

[BDWZ12] A. Burns, R.I. Davis, P. Wang, and F. Zhang. Partitioned edf scheduling for multiprocessors using a c=d scheme. *Real-Time Systems Journal, 48(1)*, pages 3–33, 2012.

[BGP95] S.K. Baruah, J.E. Gehrke, and C.G. Plaxton. Fast scheduling of periodic tasks on multiple resources. *In Proceedings of the 9th International Parallel Processing Symposium (IPPS), Santa Barbara, California*, pages 280–288, 1995.

[BHR93] S.K. Baruah, R.R. Howell, and L.E. Rosier. Feasibility problems for recurring tasks on one processor. *Theoretical Computer Science, 118(1)*, pages 3–20, 1993.

[Bim07] F. Bimbard. Dimensionnement temporel de systèmes embarqués : Application à osek. 2007.

[BKM+91] S. Baruah, G. Koren, B. Mishra, D. Mao, A. Raghunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. *In Proceedings of the 12th Real-Time Systems Symposium*, pages 106–115, 1991.

[BLOS95] A. Burchard, J. Liebeherr, Y. Oh, and S. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers, 44(12)*, 1995.

[BRH90] S.-K. Baruah, L.-E. Rosier, and R.-R. Howell. Algorithms and complexity : Concerning the preemptive scheduling of periodic real-time tasks on one processor. *Real-Time Systems Journal, 2(4)*, pages 301–324, 1990.

[BS93] G. Buttazzo and J. Stankovic. RED : Robust Earliest Deadline scheduling. *In Proceedings of the International Workshop on Responsive Computing Systems*, pages 319–355, 1993.

[BSS95] G. Buttazzo, M. Spuri, and F. Sensini. Value vs. deadline scheduling in overload conditions. *In Proceedings of the 15th Real-Time System Symposium*, pages 90–99, 1995.

[But97] G.-C. Buttazzo. Hard real-time computing systems. *Kluwer Academic Press*, 1997.

[CB97] M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsivess. *In Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, 1997.

[CB98] M. Caccamo and G. Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems. *In Proceedings of the IEEE Real-Time Computing Systems and Applications*, 1998.

[CDKM00] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. Ordonnancement temps réel - cours et exercices corrigés. *Hermès*, 2000.

[CGMV99] E.G. Coffman, G. Galambos, S. Martello, and D. Vigo. Bin packing approximation algorithms : Combinatorial analysis. *Handbook of Combinatorial Optimization, Kluwer Academic Publishers, Boston, MA*, pages 151–207, 1999.

[CHB79] R.-H. Campbell, K.-H. Horton, and G.-G. Belford. Simulations of a fault-tolerant deadline mechanism. *Digest of papers FTCS-9*, pages 95–101, 1979.

[Che13] M. Chetto. Scheduling periodic tasks with energy harvesting constraints. *Technical Report, IRCCyN, University of Nantes*, 2013.

[CLL90] J.-Y. Chung, J.-W.-S. Liu, and K. Lin. Scheduling periodic jobs that allow imprecise results. *In IEEE Transactions on Computers, 39(9)*, pages 1156–1174, 1990.

[CM96] C. Cardeira and Z. Mammeri. Neural network versus max-flow algorithms for multiprocessor real-time scheduling. *In Proceedings of the IEEE EURWRTS*, pages 175–180, 1996.

[Cof76] E.G. Coffman. Introduction to deterministic scheduling theory. *Computer and Job-Shop Scheduling Theory*, 1976.

[CSM97] C. Cardeira, M. Silva, and Z. Mammeri. Handling precedence constraints with neural network based real-time scheduling algorithms. *In Proceedings of the 9th Euromicro Workshop on Real-Time Systems*, pages 207–214, 1997.

[CT10] J.-J. Chen and L. Thiele. Energy-efficient scheduling on homogeneous multiprocessor platforms. *In Proceedings of the 25th ACM Symposium on Applied Computing, Switzerland*, 2010.

[DB11] R.I. Davis and A. Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. *ACM Computing Surveys (CSUR), 2011*, 2011.

[Del94] J. Delacroix. Un contrôleur d'ordonnancement temps-réel pour la stabilité de earliest deadline en surcharge : le régisseur. *PhD Thesis, Université Pierre et Marie Curie*, 1994.

[Der74] M.-L. Dertouzos. Control robotics : The procedural control of physical processes. *In Proceedings of the International Federation for Information Processing Congress*, pages 807–813, 1974.

[DL78] S.K. Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research, 26(1)*, pages 127–140, 1978.

[DM89] M.L. Dertouzos and A.K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering, 15(12)*, pages 1497–1506, 1989.

[DP06] K. Danne and M. Platzner. An edf schedulability test for periodic tasks on reconfigurable hardware devices. *SIGPLAN Not., 41(7)*, pages 93–102, 2006.

[DW95] R. Davis and A. Wellings. Dual priority scheduling. *In Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95)*, pages 100–109, 1995.

[GJ90] M. R. Garey and D.S. Johnson. Computers and intractability ; a guide to the theory of np-completeness. *W. H. Freeman & Co., New York, NY, USA*, 1990.

[GMR95] L. George, P. Muhlethaler, and N. Rivierre. Optimality and non-preemptive real-time scheduling revisited. *Research Report RR-2516, INRIA, Le Chesnay Cedex, France*, 1995.

[GRS96] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uniprocessor scheduling. *Research Report RR-2966, INRIA, Projet REFLECS*, 1996.

[GSYY10] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-priority multiprocessor scheduling with liu and layland's utilization bound. *In Proceedings of the 16th Real-Time and Embedded Technology and Applications Symposium*, pages 165–174, 2010.

[HLLL03] H.-S. Hu, D. Liu, M.-D. Lemmon, and Q. Ling. Firm real-time system scheduling based on a novel QoS constraint. *In Proceedings of the 24th Real-Time Systems Symposium (RTSS'03)*, 2003.

[Hol04] A. S. Holmes. Axial-flow microturbine with electromagnetic generator : Design, cfd simulation, and prototype demonstration. *In Proceedings of 17th IEEE International Micro Electro Mechanical Systems Conf. (MEMS 04)*, pages 568–571, 2004.

[Hor74] W.A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly 21*, pages 177–185, 1974.

[HR95] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers, 44(4)*, pages 1443–1451, 1995.

[HT85] J. J. HOPFIELD and D. W. TANK. Neural computation of decisions in optimization problems. *Biological Cybernetics, 52*, pages 141–52, 1985.

[Jac55] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. *Research Report Number 43, Management Science Research Project, UCLA*, 1955.

[JG99] K. Jeffay and S. Godard. A theory of rate-based execution. *In Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS'99)*, pages 304–314, 1999.

[JNC+89] E.-D. Jensen, J.-D. Northcutt, R.-K. Clark, S.-E. Shipman, F.-D. Reynolds, D.-P. Maynard, and K.-P. Loepfere. Alpha : An operating system for the mission-critical integration and operation of large, complex, distributed real-time systems. *In Proceedings of the 1989 Workshop on Mission Critical Operating Systems*, 1989.

[JP86] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal, 29(5)*, pages 390–395, 1986.

[KGJ03] P. Kohout, B. Ganesh, and B. Jacob. Hardware support for real-time operating systems. *In Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 45–51, 2003.

[KS93] G. Koren and D. Shasha. d^{over} : An optimal online scheduling algorithm for overloaded realtime systems. *In Proceedings of the IEEE Real-Time Systems Symposium*, pages 290–299, 1993.

[KS95] G. Koren and D. Shasha. Skip-over : Algorithms and complexity for overloaded real-time systems. *In Proceedings of the IEEE Real Time Systems*, 1995.

[KSMI03] P. Kuacharoen, M. Shalan, and V. Mooney III. A configurable hardware scheduler for real-time systems. *In Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms*, pages 96–101, 2003.

[KY07] S. Kato and N. Yamasaki. Real-time scheduling with task splitting on multiprocessors. *In Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 441–450, 2007.

[KY08a] S. Kato and N. Yamasaki. Portioned edf-based scheduling on multiprocessors. *In Proceedings of the 8th ACM International Conference on Embedded Software*, pages 139–148, 2008.

[KY08b] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. *In Proceedings of the International Symposium on Parallel and Distributed Processing*, pages 1–12, 2008.

[KY08c] S. Kato and N. Yamasaki. Semi-partitioning technique for multiprocessor real-time scheduling. *In Proceedings of the 29th Real-Time Systems Symposium*, 2008.

[KY09] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. *In Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 23–32, 2009.

[KYI09] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. *In Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 249–258, 2009.

[LDG01] J.M. Lopez, J.L. Diaz, and D.F. Garcia. Minimum and maximum utilization bounds for multiprocessor rm scheduling. *In Proceedings of the 13th Euromicro Conference on Real-Time Systems*, pages 67–75, 2001.

[LGDG00] J.M. Lopez, M. Garcia, J. Diaz, and D.F. Garcia. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. *In Proceedings of the Euromicro Conference on Real-time Systems*, pages 25–33, 2000.

[Liu00] J. W. S. W. Liu. Real-time systems. *Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition*, 2000.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM, 20(1)*, pages 46–61, 1973.

[LLN87] J.-W.-S. Liu, J.-K. Lin, and S. Natarajan. Scheduling algorithms for multiprogramming in a hard real-time environment. *In Proceedings of the 8th Real-Time System Symposium, San Francisco, USA*, pages 252–260, 1987.

[LM80] J. Y.-T. Leung and M.L. Merill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters, 11(3)*, pages 115–118, 1980.

[LNL87] J.-K. Lin, S. Natarajan, and J.-W.-S. Liu. Condor : A distributed system making use of imprecise results. *In Proceedings of COMPSAC'87, Tokyo, Japan*, 1987.

[Loc86] C.-D. Locke. Best-effort decision making for real-time scheduling. *PhD Thesis, Computer Science Department, Carnegie-Mellon University*, 1986.

[LQ11] J. Lu and Q. Qiu. Scheduling and mapping of periodic tasks on multi-core embedded systems with energy harvesting. *Journal of Computers and Electrical Engineering*, pages 498–510, 2011.

[LQW08] S. Liu, Q. Qiu, and Q. Wu. Energy aware dynamic voltage and frequency selection for real-time systems with energy harvesting. *In Proceedings of Design, Automation, and Test in Europe*, 2008.

[LQW09] S. Liu, Q. Qiu, and Q. Wu. An adaptive scheduling and voltage/frequency selection algorithm for real-time energy harvesting systems. *In Proceedings of Design Automation Conference*, 2009.

[LRL09] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multicore processors. *In Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 239–248, 2009.

[LSD89] J.-P. Lehozcky, L. Sha, and Y. Ding. The rate-monotonic scheduling algorithm : exact characterization and average case behaviour. *In Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[LW82] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation 2*, pages 237–250, 1982.

[MBTB06] C. Moser, D. Brunelli, L. Thiele, and L. Benini. Lazy scheduling for energy-harvesting sensor nodes. *In Proceedings of 5th Working Conference on Distributed and Parallel Embedded Systems*, 2006.

[MD78] A.-K. Mok and M.-L. Dertouzos. Multiprocessor scheduling in a hard real-time environmnent. *In Proceedings of the 7th Texas Conference on Computer Systems*, 1978.

[MF02] C. Montez and J. Fraga. Dealing with overloading in tasks scheduling. *In Proceedings of the 12th International Conference of the Clilean Computer Science Society (SCCC'02), Copiapo, Atacama, Chile*, 2002.

[MGYH04] P. D. Mitcheson, T. C. Green, E. M. Yeatman, and A. S. Holmes. Analysis of optimized micro-generator architectures for self-powered ubiquitous computers. *Imperial College of Science Technology and Medicine. Exhibition Road, London, SW7 2BT*, 2004.

[ML04] A. Morton and W.M. Loucks. A hardware/software kernel for system on chip designs. *In Proceedings of the ACM Symposium on Applied Computing*, pages 869–875, 2004.

[Mok83] A.K. Mok. Fundamental design problems of distributed systems for the hard real-time environments. *PhD thesis, MIT, Dept. of Electrical Engineering and Computer Science*, 1983.

[MS95] M. Marucheck and J. Strosnider. An evaluation of the graceful degradation properties of real-time schedulers. *In Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing*, 1995.

[OS95] Y. Oh and S. Sang. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems Journal, 9(3)*, pages 207–239, 1995.

[PF01] J. Paradiso and M. Feldmeier. A compact, wireless, self-powered pushbutton controller. *ubi-comp : Ubiquitous Computing*, 2001.

[PSTW97] C.A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *In Proceedings of the ACM Symposium on theory of Computing*, pages 42–48, 1997.

[Que06] A. Queudet. Ordonnancement temps réel avec contraintes de qualite de service. *Thèse de Doctorat de l'Université de Nantes*, 2006.

[RADS+00] J. M. Rabaey, M. J. Ammer, J. L. Da Silva, D. Patel, and S. Roundy. Picoradio supports ad hoc ultra-low power wireless networking. *IEEE Computer*, pages 42–48, 2000.

[RRC05] F. Ridouard, P. Richard, and F. Cottet. Ordonnancement de tâches indépendantes avec suspension. *In Proceedings of the 13th International Conference on Real-Time Systems*, 2005.

[RSF+04] S. Roundy, D. Steingart, L. Fréchette, P. K. Wright, and J. Rabaey. Power sources for wireless networks. *In Proceedings of the 1st European Workshop on Wireless Sensor Networks (EWSN '04), Berlin, Germany*, 2004.

[RWP02] S. Roundy, P. K. Wright, and K. S. Pister. Micro-electrostatic vibration-to-electricity converters. *In Proceedings of the ASME International Mechanical Engineering Congress and Expo*, 2002.

[SP01] N. S. Shenck and J. A. Paradiso. Energy scavenging with shoe-mounted piezoelectrics. *IEEE Micro, 20*, pages 30–41, 2001.

[SP04] T. Starner and J. A. Paradiso. Human-generated power for mobile electronics. *In C. Piguet (Ed). Low-power electronics design, chapter 45*, pages 1–35, 2004.

[Sta88] J. Stankovic. Misconceptions about real-time computing. *IEEE Computer*, 1988.

[Sta94] J. Stankovic. Adjustable flow control filters and reflective memories as support for distributed real-time systems. *Technical Report UM-CS-1994-034, University of Massachusetts, Amherst, Computer Science*, 1994.

[Ste99] J. Stevens. Optimized thermal design of small thermoelectric generators. *In Proceedings of 34th Intersociety Energy Conversion Eng. Conference. Society of Automotive Engineers*, 1999.

[SVC98] S. Saez, J. Vila, and A. Crespo. Using exact feasibility tests for allocating real-time tasks in multiprocessor systems. *In Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 53–60, 1998.

[WGCH10] T. Wei, Y. Guo, X. Chen, and S. Hu. Adaptive task allocation for multiprocessor SoCs. *In Proceedings of the 11th International Symposium on Quality Electronic Design*, 2010.

[WGS01] R. West, I. Ganev, and K. Schwan. Window-constrained process scheduling for linux systems. *In Proceedings of the 3rd Real-Time Linux Workshop, Milan, Italy*, 2001.

[WS99] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. *In Proceedings of the 6th IEEE International Conference on Multimedia Computing and Systems (ICMCS'99)*, 1999.

[Yag02] O. Yaglioglu. Modeling and design considerations for a micro-hydraulic piezoelectric power generator. *Master's thesis, Department of Electrical Eng. and Computer Science, MIT*, 2002.

[Yea04] E.M. Yeatman. Advances in power sources for wireless sensor nodes. *In Proceedings of the International Workshop on Wearable and Implantable Body Sensor Networks, Imperial College*, pages 20–21, 2004.

[Yil11] F. Yildiz. Potential ambient energy-harvesting sources and techniques. *The Journal of Technology Studies*, 2011.

[ZB08a] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *Technical Report YCS 426, University of York*, 2008.

[ZB08b] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *In Proceedings of the IEEE Transaction on Computers, 58(9)*, pages 1250–1258, 2008.

[ZYTT09] G. Zeng, T. Yokoyama, H. Tomiyama, and H. Takada. Practical energy-aware scheduling for real-time multiprocessor systems. *In Proceedings of IEEE International Conference on Embedded and Real-time Computing Systems and Applications*, 2009.

l'université nantes angers le mans
PÔLE DE RECHERCHE ET D'ENSEIGNEMENT SUPÉRIEUR

UNIVERSITÉ DE NANTES

Thèse de Doctorat

Nadine ABDALLAH

Partitionnement temps réel multiprocesseur sous contraintes de qualité de service et d'énergie

Multiprocessor real-time partitioning with Quality of Service requirements and energy constraints

Résumé

Ces travaux de thèse s'inscrivent dans le domaine de l'informatique temps réel multiprocesseur contrainte par l'énergie. Ils visent à proposer des stratégies de partitionnement de tâches sur une plateforme constituée de processeurs identiques puis à mesurer leur performance par une étude de simulation. Ces stratégies se singularisent par leur flexibilité et facilité d'implémentation et ce, pour différents modèles de tâches : (i) à contraintes strictes/fermes, (ii) à échéances contraintes/à échéances sur requêtes, (iii) synchrones/asynchrones. La première partie de ces travaux vise à l'amélioration et la proposition de nouvelles stratégies de partitionnement pour des systèmes non contraint par l'énergie. La seconde partie se focalise sur les systèmes temps réel embarqués autonomes alimentés par l'énergie tirée de l'environnement (energy harvesting). Le modèle étudié considère que chaque processeur dispose de son propre réservoir d'énergie de taille donnée et que la puissance de la source environnementale ne varie pas au cours du temps. La validation par simulation des heuristiques de partitionnement proposées montre qu'il est possible d'atteindre un niveau de performance similaire à celui des approches de semi-partitionnement tout en s'affranchissant de leurs inconvénients, notamment des coûts de migration. Nous accompagnons ces stratégies d'un guide de choix en vue d'aider un concepteur dans la difficile tâche du dimensionnement de son système caractérisé par les réservoirs de stockage (batterie ou supercondensateur) et les récupérateurs d'énergie.

Mots clés

partitionnement, systèmes temps réel embarqués autonomes énergétiquement, multiprocesseur, qualité de service.

Abstract

Thesis work fits in the area of multiprocessor real-time computing under energy constraints. It aims to propose partitioning strategies for tasks on a homogeneous multiprocessor platform and to measure their performance through simulations. These strategies stand out because of their flexibility and ease of implementation for different task models: (i) hard/firm real-time constraints, (ii) implicit/constrained deadlines, (iii) synchronous/asynchronous. First, we extend and propose new partitioning strategies for systems without energy considerations. The second part of the work focuses on energy harvesting real-time embedded systems saving energy from the environment. We assume that in the studied system model each processor of the platform has its own energy reservoir with given size and that the energy source module power does not vary over time. Simulation results show that the proposed partitioning heuristics exhibit similar performances as those that can be observed with semi-partitioning approaches without their drawbacks, in particular the migration costs. These strategies have been derived with a guide for the system helping the designer in the difficult task of choosing the best dimensioning of the system characterized by its energy storage units (battery or supercapacitor) and energy harvesting modules.

Key Words

partitioning, energy harvesting real-time embedded systems, multiprocessor, quality of service.