UNIVERSITÉ D'ORAN 1

Endormissement blabla

Auteur :

Ismail YEMMI

Encadrants :

Dr. Abou El Hassan Benyamina

Dr. Houssam-Eddine ZAHAF

Pour l'Obtention du Diplome

de Master en

INFORMATIQUE

22 Juin 2017

Nom de Jury	Grade, Université, Belgium, Pays	Fonction
Nom de Jury	Grade, Université, Belgium, Pays	Fonction
Nom de Jury	Grade, Université, Belgium, Pays	Fonction
Nom de Jury	Grade, Université, Belgium, Pays	Fonction

Résumé

by Ismail YEMMI

Table des matières

Table des matières

Table des figures

Liste des tableaux

List of Symbols

$\%$	The rest of the ecludienne division
$\mathsf{random}(a, b)$	generates a random number between a and b
τ_i	Task i
$\mathcal{J}_{i,a}$	The a^{th} job of task τ_i
$\mathcal{A}_{i,a}$	The arrival time of a^{th} instance of task τ_i
D_i	The relative deadline of task τ_i
O_i	The offset of task τ_i
T_i	The period of task τ_i
$\mathcal{T}$	A task set
$\mathcal{T}_j$	The task set allocated on core j
$\mathcal{A}$	A multicore architecture
C_i	Execution time of the single thread version of task τ_i
$\mathcal{C}$	The execution time of an arbitrary thread
$\mathsf{tdbf}(\tau_i, t)$	The demand bound function of task τ_i for an interval of time of length t
$\mathsf{dbf}(\mathcal{T}_j, t)$	The demand bound function of task set $\mathcal{T}_j$ for an interval of time of length t
dbf	The demand bound function
α	The symbol of excess time in equations
$\vec{\xi}$	Energy coeficients

Introduction

Première partie

Partie Théorique

Chapitre 1

Introduction Au systeme temps réel

Sommaire

1.1 Introduction

Un système temps réel est capable de contrôler (ou piloter) un procédé physique afin d'assurer la correction de son fonctionnement en fonction de l'évolution de l'environnement qui l'entour. Les systèmes temps réel doivent etre capable de prendre en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat, autrement dit le système ne doit pas simplement délivrer des résultats exacts, il doit les

délivrer dans des délais impartis. Dans ce chapitre nous présentons un survol des systèmes temps réel : leurs leur modélisation (modèles de taches, ordonanncements), leur implantation, $\cdots$ sur des plateformes monoprocesseurs et multiprocesseurs.

1.2 Taxonomie sur les systèmes temps réel

Différents niveaux de criticité

Les systèmes temps réel *critiques* (ou *dur*) correspondent aux systèmes pour lesquelles il est intolérable qu'une contraintes temps réel soitent violée et la violation de cette derniere risque de causer des conséquences graves telles que des blessures ou des pertes humaines. Les centrales nucléaires ou le guidage de missiles représentent de tels systèmes à haute criticité. Dans le domaine de l'informatique embarqué, l'automobile et l'aéronautique regorgent de systèmes critiques à l'image des équipements déclencheurs d'airbags ou des logiciels de contrôle de vol de satellite. Il est crucial que les résultats soient disponibles au moment voulu et un résultat obtenu trop tard est inutilisable, à l'instar d'un système anti-missile qui recevrait la position d'un objet volant avec du retard.

Les systèmes temps réels mou sont des systèmes où on tolère les retards et ne requièrent pas un déterminisme temporel aussi fort que les systèmes temps réels dur. Par exemple, un logiciel de diffusion de flux vidéo produit un certain nombre d'images dans un intervalle de temps régulier. Le fait de manquer une ou plusieurs échéances ne provoque pas l'arrêt du système multimédia. La qualité de la vidéo est dégradée mais le service peut continuer de fonctionner sans risque. Donc les systèmes temps réels mou offre le meilleur service possible (notion de best effort) et les retards dans l'obtention des résultats ne sont pas dramatiques.

A la frontière entre les systèmes temps réel dur et mou, les systèmes temps réel ferme tolèrent une certaine proportion d'échéances manquées. Ils ne considèrent que les résultats obtenus à temps et sont liés à la notion de qualité de service (QoS).

1.3 Modélisation des tâches

Les systèmes temps réel sont des systèmes qui interagissent avec l'environnement. Ainsi, les fonctionnalités d'un système temps réel sont réccurentes, et peuvent s'executer durant tout la vie du système. Liu et Layland **LL73** sont les premiers à proposer une modelisation d'un systemes temps réel comme suit :
Soit $\mathcal{T}$ un système temps réel composé d'un ensemble de n tâches (fonctionnalitées). Les tâches sont périodiques, et leurs caractéristiques sont détaillées ci-dessous.

Tâche : Une tâche τ est définie comme l'exécution d'une suite d'instructions. Nous supposons que toutes les tâches sont indépendantes et que l'ordre dans lequel les tâches sont exécutées n'a pas de conséquence sur la bonne exécution du système du moment qu'elles respectent leurs contraintes temporelles. Nous faisons également l'hypothèse que les tâches sont synchrones, donc que toutes les tâches sont actives dès que le système débute son exécution, les tâches sont toutes libérées simultanément.

Travail (Job) : Chaque tâche libère périodiquement des travaux. Un travail est une suite d'instructions qui doit être réalisée avant une date fixée. Lorsqu'une tâche libère un travail, celui-ci est prêt à être exécuté et devient disponible pour l'algorithme d'ordonnancement. Une tâche τ_i libère ses travaux périodiquement suivant sa période T_i, un travail n'a donc pas de période associée. Ce modèle est appelé modèle de tâche périodique car chaque travail est libéré exactement lorsque la tâche atteint sa période. D'autres modélisations plus souples existent comme les systèmes de tâches sporadiques ou apériodiques. Pour les systèmes sporadiques, la période d'une tâche est la période de temps minimale entre deux libérations de travaux pour une tâche, ce qui signifie que le système ne peut savoir la date exacte où le travail va être libéré. Dans le cas de systèmes apériodiques, l'intervalle de temps entre deux libérations de travaux n'est soumis à aucune contrainte. Ces systèmes sont plus difficiles à étudier du fait de l'imprévisibilité de l'arrivée des tâches.

Hyperpériode : L'hyperpériode H correspond au plus petit commun multiple de toutes les périodes de l'ensemble de tâches. $H = PPCM(\{\mathsf{T}_0, \mathsf{T}_1, \cdots, \mathsf{T}_n\})$.

Date Réveil : notée O_i, c'est la date où la tâche libère son premier travail, chaque travail de la tâche est libéré à l'instant $\mathsf{O}_i + K \cdot \mathsf{T}_i,\ K \in \mathbb{N}$

Pire temps d'execution (Worst Case Execution Time WCET, C_i) : est la durée maximale de l'exécution de chacun de ses travaux. Le WCET de la tâche τ_i est noté C_i. Calculer le WCET d'une tâche est difficile et ce sujet est une thématique de recherche à lui tout seul. Nous renvoyons le lecteur à **WEE+08** pour plus d'informations. Nous supposons que le WCET de chaque tâche est connu à priori.

Échéance(Deadline) : Chaque travail une fois libéré doit terminer son exécution avec une certaine date sous peine de violer son échéance. Nous notons D_i l'échéance relative de la tâche τ_i. L'échéance absolue $d_{i,j}$ du travail j sera donc la date de sa libération additionnée de cette échéance relative.

Selon l'échéance et la période, il existe trois types de modèles de tâches :

— Modèles à *échéances implicites* où l'échéance de chaque travail égale à sa période $\mathsf{T}_i = \mathsf{D}_i$.

— Modèles à *échéances contraintes* où l'échéance de chaque travail est inférieure ou égale à sa période $\mathsf{D}_i \leq \mathsf{T}_i$.

— le modèle à *échéances arbitraires* ne fixe aucune contrainte entre les échéances et les périodes des tâches.

Utilisation d'une tâche : L'utilisation d'une tâche est le rapport entre son pire temps d'exécution C_i et sa période T_i. L'utilisation u_i de la tâche τ_i est donc $\frac{\mathsf{C}_i}{\mathsf{T}_i}$

Utilisation globale de l'ensemble de tâches : L'utilisation globale U de l'ensemble de tâches est la somme de toutes les utilisations individuelles des tâches de l'ensemble de tâches :

$$U = \sum_{i=1}^{n} u_i \tag{1.1}$$

1.4 Ordonnancement monoprocesseur

L'ensemble des fonctionnalité d'un système temps réel sont en concurrence sur les différents ressources. Afin de gérer les ressources partagées des algorithmes "d'arbitrage" de ressources partagées a besoin d'etre mis en oeuvre. Pour la ressource *processeur*, ces algorithmes sont appelés particulierement "ordonnanceurs". Un algorithme d'ordonnancement monoprocesseur est chargé de décider quelle tâche sera exécutée sur le processeur et "probablement" pour combien de temps.

Definition

Dans un premier temps, nous définissons les termes habituels concernant l'ordonnancement temps réel

Hors-ligne / en-ligne : Un algorithme d'ordonnancement hors-ligne prend la totalité de ses décisions d'ordonnancement avant l'exécution du système. Au contraire, un ordonnancement en-ligne prend les décisions d'ordonnancement lors de l'exécution

Priorités : Les algorithmes d'ordonnancement temps réel peuvent être classés suivant leur utilisation des priorités pour choisir quelle tâche doit être ordonnancée.

Préemptif / non préemptif : Un algorithme d'ordonnancement préemptif est un algorithme d'ordonnancement qui peut arrêter l'exécution d'une tâche à tout moment lors de l'exécution. Au contraire, un algorithme d'ordonnancement non préemptif ne permet aucune préemption, un travail en cours d'exécution ne peut être arrêté et le processeur ne peut etre utilisé par une autre tache qu'à la fin de l'execution de ce dernier.

Ordonnançabilité / Faisabilité : Un système de tâches est dit ordonnançable si un ordonnancement existe permettant de satisfaire toutes les contraintes temps réel. Un système de tâches est dit faisable s'il existe un algorithme d'ordonnancement permettant d'ordonnancer ce système de tâches sans aucune violation d'échéances.

Optimalité : Un algorithme d'ordonnancement est dit optimal s'il peut ordonnancer tous les ensembles de tâches ordonnançables par d'autres algorithmes d'ordonnancement existants.

Test d'ordo

Test suffisasnt

Test necessaire

Test Exacte

[**Houssam**: Ajoute les définitions qui faut ci-dessus]

Selon la priorité, les algorithmes d'ordonnancement sont classés comme ci-dessous :

!

1.4.1 Algorithme d'ordonnancement à priorité fixe

Rate Monotonic LL73

Rate Monotonic est un algorithme à priorité fixe introduit par Liu et Layland dans **LL73** Cet algorithme affecte les priorités aux tâches inversement proportionnel à leur période : plus leur période est petite, plus la tâche est prioritaire.

Un exemple de système de tâche ordonnancée par Rate Monotonic est donné table 1.1. La figure **??** est une représentation graphique de l'ordonnancement correspondant.

τ_i	C_i	T_i	priorité
1	1	10	3
2	1	4	0
3	1	5	1
4	2	8	2

TABLE 1.1 – ensemble de tache avec priorité affecté par Rate Monotonic

Theorème 1. *Rate Monotonic est optimal pour l'ordonnancement de systèmes de tâches synchrones, indépendantes et à échéance sur requête en présence de préemption.*

Theorème 2 (Condition Suffisante **LL73).** *. Un système temps réel composé de n tâches est ordonnançable par Rate Monotonic si :*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{1.2}$$

Deadline Monotonic LW82

Deadline Monotonic est un algorithme à priorité fixe introduit par Leung et Whitehead dans **LW82**

Cet algorithme est proche de celui de Rate Monotonic, à la différence que les priorités sont maintenant affectées en fonction de l'échéance relative de chaque tâche au lieu de leur période.

Theorème 3. *Cet algorithme est optimal dans le cadre des algorithmes à priorité fixe pour des systèmes de tâches synchrones à échéance contrainte lorsque la préemption est autorisée. Monotonic et Deadline Monotonic se confondent.*

Condition suffisante d'ordonnançabilité La condition suffisante d'ordonnançabilité est inspirée de la condition suffisante d'ordonnançabilité de Liu et Layland (cf. théorème 4) :

Theorème 4. *Un système temps réel composé de n tâches est ordonnançable par Deadline Monotonic si la condition suivante est vérifiée :*

$$U = \sum_{i=1}^{n} \frac{\mathsf{C}_i}{\mathsf{D}_i} \leq n(2^{\frac{1}{n}} - 1) \tag{1.3}$$

Condition necessaire et suffisante d'ordonnançabilité Joseph et al. **JP86** ont proposé un test d'ordonnancabilité basé sur le pire temps de reponse R_i. Le pire temps de réponse est le moment ou la tache τ_i de priorité p_i terminera son exécution quand elle et toutes les tâches les plus prioritaire sont actifs même temps.

Theorème 5. *soit $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$ un ensemble de n tâches. $\mathcal{T}$ est ordonnançable sous deadline monotonic si et seulement si :*

$$\forall \tau_i \in \mathcal{T}, R_i \leq \mathsf{D}_i \tag{1.4}$$

$$R_i^0 = \mathsf{C}_i \tag{1.5}$$

$$R_i^{(k+1)} = \mathsf{C}_i + \sum_{j \in pr(i)} \left\lceil \frac{R_i^{(k)}}{\mathsf{T}_j} \right\rceil \cdot \mathsf{C}_j \tag{1.6}$$

τ_i	C_i	D_i	T_i	Pr_i	R_i
1	1	5	10	2	3
2	1	3	4	0	1
3	1	4	5	1	2
4	2	7	8	3	7

TABLE 1.2 – ensemble de tache avec priorité affecté par Deadline Monotonic

1.4.2 Algorithme d'ordonnancement à priorité dynamique au tâches

Les algorithmes à priorité dynamique modifie lors de l'execution d'une tache sa priorité qui est donc plus une donnée statique. La priorité d'une tâche est mise à jour durant la vie du système en fonction de certains critères, les critères utilisés dépendant de l'algorithme utilisé. Cette catégorie d'algorithme se résume principallement à l'algorithme Earliest Deadline First (EDF).

Earliest Deadline FirstLL73

Earliest Deadline First est un algorithme un des algorithmes les plus connus et étudiés depuis longtemps **LL73** ; **Der74** ; **Hor74** Le principe de cet algorithme est d'accorder la priorité la plus grande à la tâche ayant une instance dont l'échéance absolue est la plus proche. L'avantage majeur de cet algorithme est qu'en présence d'un système de tâche à échéance sur requête, le taux d'utilisation maximum du processeur est de 100% (théorème 6). [**Houssam**: STP prends soin à copier les choses correctement, ce théoreme était totalement faux.]

Theorème 6 (LL73). *Un système de n tâches à échéance sur requete est ordonnançable par Earliest Deadline First si et seulement si :*

$$U \leq 1 \tag{1.7}$$

[**Houssam**: Encore une fois une définition fausse ! ! ! il faut vérifier les sources]

Fonction de demande du processeur La demande du processeur des tâches est une fonction qui calcul la demandes maximale de taches qui ont leur activation et échéance

dans n'importe quel interval de temps de longueur t notée $\mathsf{dbf}(\mathcal{T}, t)$. Durant le reste du mémoire, nous ferons référence à la fonction de demande du processeur par dbf.

$$\mathsf{dbf}(\mathcal{T}, t) = \sum_{\tau_i \in \mathcal{T}} \left\lfloor \frac{t - \mathsf{T}_i + \mathsf{D}_i}{\mathsf{T}_i} \right\rfloor \mathsf{C}_i \tag{1.8}$$

Condition necessaire et suffisante d'ordonnançabilité BHR93

il existe un test d'ordonnancabilité basée sur la fonction de la demande processeur $\mathsf{dbf}(\mathcal{T}, t)$ causée par des tâches activées et devant être terminées dans l'intervalle de longueur t.

Theorème 7. *Un système $\mathcal{T}$ de n tâches à échéance contrainte est ordonnançable par Earliest Deadline First si et seulement si :*

$$\forall t \geq 0, \mathsf{dbf}(\mathcal{T}, t) \leq t \tag{1.9}$$

Theorème 8 (Der74). *Earliest Deadline First est optimal pour ordonnancer des systèmes de tâches indépendantes lorsque le facteur d'utilisation U du système est inférieur ou égale à 1 (absence de surcharge).*

τ_i	C_i	D_i	T_i
1	3	4	5
2	3	7	7

TABLE 1.3 – ensemble de tache

[**Houssam**: Sur tout tes examples il faut ajouter une description des exemples]

1.5 Ordonnancement Multiprocesseur

!

L'ordonnancement multiprocesseur se distingue de l'ordonnancement monoprocesseur par la présence de plusieurs processeurs sur lesquels peuvent s'exécuter les tâches. Se pose alors certains problèmes parmis eux :

— le problème de placement des tâches : sur quel(s) processeur(s) une tâche va-t-elle s'exécuter ?

— le problème de la migration des tâches : une tâche peut-elle changer de processeur pour s'exécuter ?

— le problème de l'ordonnancement des tâches : affectation des priorités.

1.5.1 Classification

Les algorithmes d'ordonnancement peuvent être classés dans différentes catégories, en fonction de leurs caractéristiques :

— stratégie globale : sur un système comprenant m processeurs, un algorithme d'ordonnancement utilisant une stratégie globale va affecter les m tâches les plus prioritaires aux m processeurs.

— stratégie partitionnée : le principe d'un algorithme utilisant une stratégie par partitionnement est de placer chaque tâche sur un processeur, et ensuite d'exécuter sur chaque processeur un algorithme d'ordonnancement monoprocesseur. Comme vous pouvez le remarquer dans une stratégie partitionée une tache s'execute seulement sur un processeur au contraire de la stratégie globale ou les taches ont le droit de s'executer sur différents processeurs.

— stratégie semi partitionnée : Elle est obtenue par combinaison de la stratégie par partitionnement et de la stratégie globale. Ainsi, les taches sont alloués à des processeurs et certain jotravaux de certaines taches ont le droit de migrer à un autre processeur.

Il est à noter qu'il n'y a pas une catégorie qui soit meilleure qu'une autre. Il existe des systèmes de tâches qui peuvent être ordonnancés en utilisant une stratégie globale mais pas avec une stratégie par partitionnement et inversement. On dit que ces algorithmes sont **incomparables LW82**

1.5.2 Optimalité

Theorème 9 (HL92). *Il n'existe pas d'algorithme en-ligne optimal pour des systèmes multiprocesseurs. Toutefois, lorsqu'on restreint le cadre d'étude et que l'on ne considère uniquement des*

systèmes de tâches périodiques, à échéance sur requete, alors il existe des algorithmes optimaux, comme les algorithmes de type Pfair par exemple.

Dans ce manuscrit, nous nous interessons aux stratégies partitionées parce qu'ils sont simple à implémenter et leur testes d'ordonnançabilités sont basés sur les testes mono-processeurs.

1.5.3 Algorithme d'ordonnancement utilisant une stratégie par partitionnement

Généralité

Les algorithmes utilisant une stratégie par partitionnement relèvent dans la plupart des cas du problème du bin-packing, c'est-à-dire comment trouver un placement pour l'ensemble des tâches sur un nombre minimum de processeurs. Ce problème de partitionnement des tâches pour les placer sur les processeurs a été montré comme étant NP-difficile dans **LW82** Il n'existe donc pas d'algorithmes s'exécutant en temps polynomial permettant de trouver une solution optimale à ce problème. Toutefois, il existe des heuristiques permettant d'obtenir des résultats corrects en temps polynomial.

First-Fit et Best-Fit

Parmi les heuristiques existantes pour résoudre ce type de problème, quatre heuristiques : First Fit, Best Fit, Next Fit et Worst Fit qui reposent tous sur le même principe : on affecte chaque tâche dans l'ordre à un processeur, selon un critère d'acceptation. La différence entre les deux types d'algorithmes réside sur la manière dont est placée chaque tâche :

First Fit : la tâche est placée sur le premier processeur avec un ensemble de tache ordonnançable ; [**Houssam**: dis que ça commence toujours par le proc 0 au contraire de NF ou ça continue !]

Next Fit : même principe que First Fit [**Houssam**: Si c'est le meme principe pourquoi il y a deux ! ! ! !] !

!

Best Fit : la tâche est placée sur le processeur avec un ensemble de tache ordonnançable et ayant le taux d'utilisation maximal.

Worst Fit : la tâche est placée sur le processeur avec un ensemble de tache ordonnançable et ayant le taux d'utilisation minimal.

[**Houssam**: Il faut dire que le teste d'acceptation doit etre un teste d'ordo]

Les algorithmes de type First Fit ou Best Fit ne sont pas les seuls existants il existe aussi Next Fit ou aussi Worst Fit. !

La table 1.4 et les figures **??** et **??** illustre les deux heuristiques de placement First-Fit et Best-Fit sur un ensemble de 10 taches periodiques et 4 processeurs.

τ_i	C_i	D_i	T_i
1	5	10	10
2	7	21	21
3	3	22	22
4	1	24	24
5	10	30	30
6	16	40	40
7	1	50	50
8	3	55	55
9	9	70	70
10	17	100	100

TABLE 1.4 – Ensemble de taches periodiques

P1	τ_1	τ_3	τ_4	τ_7	τ_8
P2	τ_2	τ_5			
P3	τ_6	τ_9	τ_{10}		
P4	$\emptyset$				

FIGURE 1.1 – Placement de taches en First Fit par les contraintes DM

[**Houssam**: Parle un petit peu des algorithmes d'ordonnancement globale !]

!

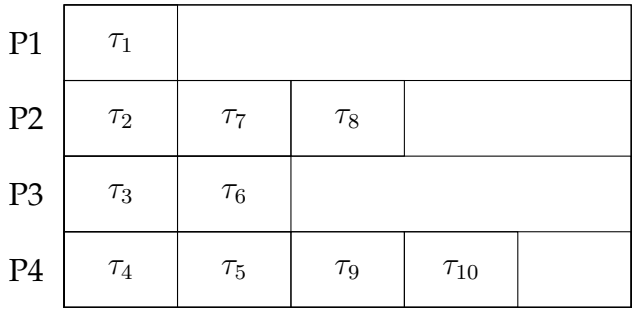

FIGURE 1.2 – Placement de taches en Best Fit par les contraintes DM

1.6 Conclusion

Dans ce chapitre, nous avons fait un survol sur les systèmes temps réel. Nous avons particulierement mis l'accent sur l'ordonanncement temps réel. Nous avons présenté tout les testes d'ordonanaçabilité qui seront utilisés dans le reste de ce document. En ce qui concerne les algorithme d'ordonnancement : Nous utiliserons DM et EDF pour le mono-processeur et un DM et EDF partitioné pour le cas multiprocesseur.

Chapitre 2

Etat de l'art

Sommaire

2.1 Introduction

2.2 Les modèles de consommation d'énergie DVFS et DPM

L'énergie consommée par un processeur est l'integrale de la puissance dissipée et du temps d'exécution. Nous utiliserons principalement la notion de consommation énergétique et non de puissance. Cette consommation énergétique est divisée en consommation statique et consommation dynamique. Nous détaillons dans cette section les modèles des consommations dynamique et statique et montrons que la consommation statique est récemment devenue plus importante que la consommation dynamique.

2.2.1 Consommation dynamique

Mei et al.**Mei13** ont défini la dissipation de puissance dynamique par un circuit CMOS comme produit de Un coefficient constant qui dépend de la technologie utilisée pour

fabriquer la puce, par la Carré de la tension et par la fréquence.

$$\mathcal{P}_{dynamique} = \epsilon * \mathsf{f} * \mathsf{V}^2 \tag{2.1}$$

Où ϵ correspond à la capacité de sortie du circuit et V à la tension d'alimentation.

Les solutions permettant de réduire la puissance dynamique dans les circuits s'attachent par conséquent à réduire la fréquence de fonctionnement. La relation précédente montre qu'il est moins économique d'un point de vue énergétique de faire fonctionner un circuit à pleine vitesse que de faire fonctionner un circuit à vitesse réduite pendant un laps de temps plus important.

La technique permettant de réduire la vitesse du système est appelée DVFS (Dynamic Voltage and Frequency Scaling) et tire parti du fait que les processeurs ont plusieurs fréquences et plusieurs tensions de fonctionnement. De nombreux algorithmes d'ordonnancement ont été proposés utilisant cette solution **WWDS94** ; **YDS95** ; **PS01**

2.2.2 Consommation statique

La consommation statique des processeurs est en grande partie due aux courants de fuite. Dans un circuit idéal, cette consommation statique est nulle mais, en réalité, un courant de fuite existe et est responsable d'une consommation énergétique non négligeable. Ce courant de fuite provient du fait que les transistors composant le circuit ne sont pas parfaits. La consommation statique peut être modélisée comme une constante **KAB+03** ; **SJPL08** Plusieurs solutions matérielles existent pour réduire la consommation statique. L'idée générale est d'éteindre une partie du circuit qui n'est pas utilisée pour qu'aucun courant de fuite ne circule. Cette solution est appelée Power Gating. Dans les circuits actuels, les puces peuvent être divisées en plusieurs parties, chaque partie ayant la possibilité d'être éteinte indépendamment des autres parties du circuit. Certaines sources **HXW+10** affirme que l'énergie statique représente jusqu'à 70% de la consommation énergétique totale dissipée par un processeur. Plusieurs études confirment cette affirmation **Bor99** ; **SBA+01** ; **KAB+03** ; **ABM+04** ; **HSC+11** ; **BBMB13** Des expériences ont également été faites **SRH05** ; **SPH05** ; **LSH10** en utilisant les algorithmes d'ordonnancement existants

et les conclusions de ces études est que réduire uniquement la consommation dynamique peut entraîner une hausse de la consommation énergétique globale car les composants sont actifs plus longtemps ce qui entraîne une hausse de la consommation statique.

2.3 Les états C-states du processeur

Pour réduire la consommation statique des processeurs, il est nécessaire d'utiliser leurs états basse-consommation lors de leurs périodes d'inactivité. Nous appelons périodes d'inactivité les périodes de temps durant lesquelles un processeur est inactif et où aucune tâche n'est exécutée. Les processeurs disposent maintenant de plusieurs états basse-consommation où un certain nombre de composants sont désactivés pour réduire la consommation énergétique. Le problème lié à l'utilisation de ces états basse-consommation est qu'éteindre, rallumer ou changer d'état un processeur n'est pas anodin, que ce soit du point de vue énergétique ou temporel. Nous définissons dans cette section quatre notions relatives aux états basse-consommation. Nous notons ns le nombre d'états basse-consommation de chaque processeur et nous supposons que tous les processeurs possèdent les mêmes états basse-consommation.

Consommation énergétique. Nous notons $Cons_s$ la consommation énergétique de l'état basse-consommation s. C'est la consommation énergétique dépensée lorsque le processeur se trouve dans cet état basse-consommation. Elle dépend du nombre de composants qui ont été désactivés. Plus ce nombre est important, plus la consommation énergétique sera réduite.

Délai de transition. Nous définissons le délai de transition d'un état basse-consommation comme le temps nécessaire pour revenir de cet état basse-consommation à l'état actif. C'est le temps nécessaire pour réactiver tous les composants éteints durant l'état basse-consommation. Le délai de transition de l'état basse-consommation s est noté Del_s. Plus la consommation énergétique de l'état basse-consommation est faible, plus son délai de transition va être important car davantage de composants devront été réactivés.

Pénalité énergétique. Nous notons Pen_s la pénalité énergétique pour revenir de l'état

basse-consommation s à l'état actif. Cette pénalité énergétique correspond à la consommation énergétique nécessaire pour réactiver tous les composants qui ont été éteints lors de l'activation de l'état basse-consommation. Elle est consommée lorsque le processeur passe de l'état basse-consommation à l'état actif, c'est-à-dire lors du délai de transition. Plus la consommation énergétique d'un état basse-consommation est faible, plus sa pénalité est importante. nous faisons l'hypothèse que l'évolution de la consommation énergétique lors du réveil du processeur est linéaire. En supposant que la consommation énergétique à l'état actif est de $Cons_{actif}$, la pénalité énergétique d'un état basse-consommation est donc donnée par la formule suivante :

$$Pen_s = \frac{1}{2} \times Del_s \times (Cons_{actif} - Cons_s) \tag{2.2}$$

Nous faisons cette hypothèse car les pénalités énergétiques des états basse-consommation sont difficiles à obtenir, les constructeurs ne fournissant généralement que les valeurs des délais de transition pour revenir des différents états basse-consommation (e.g. **STM** ; **MPC**). Néanmoins, nos contributions ne dépendent pas de cette modélisation qui n'est utilisée que pour les évaluations et qui peut être modifiée si les pénalités énergétiques des états basse-consommation sont connues.

Break Even Time (BET). Nous définissons le BET comme la largeur minimale de la pé-riode d'inactivité pour qu'il soit possible d'activer un état basse-consommation **AP11** ; **CG06** ; **DA08a** Chaque état basse-consommation possède donc son propre BET et nous nommons BET le BET de l'état basse-consommation s. Le BET de l'état basse-consommation s correspond à la période de temps pour laquelle la consommation énergétique du processeur à l'état actif est égale à la consommation énergétique dans l'état basse-consommation s (i.e. $Cons_s$) plus sa pénalité énergétique (i.e. Pen_s). Si la longueur d'une période d'inactivité est inférieure au BET d'un état basse-consommation donné, il est alors plus efficace énergétiquement de laisser le processeur dans son état actif que d'activer l'état basse-consommation en question. À noter que le BET ne permet pas de savoir si une contrainte temps réel va être violée si cet état basse-consommation est activé. C'est le délai de transition de l'état basse-consommation qui importe alors pour réveiller à temps le processeur.

Comme vu ci-dessus, nous faisons l'hypothèse que la consommation énergétique évolue linéairement pour revenir de la consommation énergétique d'un état basse-consommation à celle de l'état actif. Avec cette hypothèse, le BET et le délai de transition d'un état basse-consommation sont égaux. En d'autre termes, il est toujours plus efficace d'activer un état basse-consommation que de laisser le processeur dans l'état actif si la longueur de la période d'inactivité est supérieure au délai de transition d'un état basse-consommation. De même que pour la pénalité énergétique, nous n'utilisons cette hypothèse que dans nos évaluations et nos contributions séparent les notions de BET et de délai de transition.
Exemples de processeurs

La majorité des processeurs actuels disposent de plusieurs fréquences de fonctionnement et d'états basse-consommation pour réduire leur consommation énergétique, nous en détaillons quelques-uns ici. Comme notre objectif est de réduire la consommation statique, nous ne détaillons pas les fréquences disponibles. Nous détaillons en revanche les caractéristiques de chaque état basse-consommation, i.e. quels sont les composants éteints et quelles sont les étapes requises pour retrouver l'état actif.

Parmi les processeurs utilisés dans la littérature, citons le Freescale MPC8536 **MPC** utilisé par Awan et al. **AP11** ; **AP13** ; **AYP13** Ce processeur est basé sur une architecture PowerPC. Ses états basse-consommation sont détaillés dans le Tableau 2.4.

2.4 L'endormissement de processeur (Online VS Offline)

2.5 Le Modèle d'endormissement de Dsouza

Anthony Rowe a proposé un algorithme d'ordonnancement appellé "Energy Saving Rate Harmonized Scheduling (ES RHS)" **Rowe10** basé sur la période d'harmonisation qui améliore la consommation d'énergie pour l'architecture multiprocesseur. Dsouza l'a amelioré en proposant "ES RHS+" dans cette section nous définirons la periode harmonique, puis nous presenterons l'algorithme "Rate Harmonized Scheduling" et "Energy Saving Rate-Harmonized Scheduling" enfin nous terminerons par l'algorithme "Energy-Saving Rate-Harmonized Scheduling".

2.5.1 Période d'harmonisation

Soit un ensemble $\mathcal{T}$ de tache de n taches périodiques indépendantes, l'ensemble de tâche est ordonnée selon les périodes de chaque tache de tel façon que $\mathsf{T}_1 \leq \mathsf{T}_2 \leq \cdots \leq \mathsf{T}_n$. Un ensemble de taches $\{\tau_1, \tau_2, \ldots, \tau_n\}$ est harmonique si il existe un T_H tel que $\forall \tau_i / \frac{\mathsf{T}_i}{\mathsf{T}_H} \in \mathbb{Z}$

2.5.2 Algorithme d'ordonnancement « Energy-Saving Rate-Harmonized Scheduling »

En général RHS utilise un ensemble de Valeurs périodiques { $\mathsf{T}_H^1, \mathsf{T}_H^2, \ldots, \mathsf{T}_H^3$ } où $\mathsf{T}_H^i \leq \mathsf{T}_i$ pour $i \in \{1, \cdots, n\}$, Et les valeurs sont harmoniques. Les périodes harmoniques sont appelées "périodes de base harmonisées", Et toute les taches ont le même temps de reveil de $\tau_1 = r_1 = 0$.

Les tâches $\{\tau_1, \tau_2, \cdots, \tau_n\}$ dans l'ensemble tâches donné sont réveillées selon leurs arrivée comme dans le modèle classique Toutefois, chaque job de τ_i ne peut être exécutée que à sa prochaine frontière périodique la plus proche de T_H^i **Rowe10**

Dans le L'ordonnancement RHS de base, { $\mathsf{T}_H^1 = \mathsf{T}_H^2 = \ldots = \mathsf{T}_H^n = \mathsf{T}_H$ }, T_H Est simplement appelé période d'harmonisation et $\mathsf{T}_H \leq \mathsf{T}_1$. Les tâches qui arrivent avant ou après les multiples entiers de T_H ne sont pas autorisée à s'exécuter jusqu'à la prochaine limite la plus proche de quand ils sont lancés en fonction de leur priorité (voir figure). Les tâches qui ne sont pas admissibles sont retardées jusqu'à la prochaine limite.

T_H est choisi de manière à améliorer l'ordonnancement **Rowe10** Supposer $\Psi = \{\tau_j / \mathsf{T}_j < 2\mathsf{T}_1,\ j \neq 1\}$. si $\Psi = \emptyset, \mathsf{T}_H = \mathsf{T}_1$. *sinon* $\mathsf{T}_H = \frac{\mathsf{T}_1}{2}$.

Theorème 10 (**Rowe10**). *un ensmeble de taches $\mathcal{T}$ est faisable sous RHS si $U \leq 0.5$.*

Theorème 11. *le calcul du pire temps de reponse R_i [23,24] dans le cas de RHS peut etre redéfini comme suit :*

$$r_{i,0} = \mathsf{C}_i + \mathsf{T}_H \tag{2.3}$$

$$r_{i,k+1} = \mathsf{C}_i + \mathsf{T}_H + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^k}{\mathsf{T}_j} \right\rceil \mathsf{C}_j \tag{2.4}$$

Theorème 12 (Rowe10). *Soit $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ un ensemble de tache periodiques indépendantes.*

Si il existe une tache τ_i avec un pire temps de reponse R_i tel que $r_i > \mathsf{D}_i$ alors l'ensemble de tache $\mathcal{T}$ est non ordonnançable.

2.5.3 Energy-Saving Rate-Harmonized Scheduling

comme vu dans la section 2.3, les processeurs sont dotés de mode de basse consommation. par exemple : un processeur "power-aware" a :

— mode actif avec une consommation d'énergie PW_{active}

— mode sommeil avec une consommation d'énergie PW_{idle} et un temps de bascule ST_{idle}

— mode sommeil profond avec une consommation d'énergie PW_{sleep} et un temps de bascule ST_{sleep}

avec $PW_{active} > PW_{idle} \gg PW_{sleep}$ et $ST_{idle} \ll ST_{sleep}$.

Quand il n'y a aucune tache préte à s'executer, le processeur et generalement en mode sommeil, cependant si il y a une tache qui arrive avant $\S\mathsf{T}_{sleep}$ unité de temps,il ne peut passer en mode sommeil.

ES-RHS présente une propriété intéressante, où tous les periode de sommeil précède et est contiguë au sommeil profond durée peuvent toujours être fusionnée [10]. Par conséquent, en harmonisant les exécutions de tâches non harmoniques, ES-RHS peut produire des periode de sommeil optimal. D'souza **DAR16** a redéfini la notion d'harmonisation pour ameliorer l'ordonnancemen ES-RHS.

2.5.4 Energy-Saving Rate-Harmonized Scheduling+

Une tâche est susceptible d'être exécutée lorsque le processeur est occupé ou une limite de la période d'harmonisation a été atteinte. Cette redéfinition permet aux taches d'etre préte à s'executer plus tôt par rapport à l'ordonnanceur de ES-RHS **Rowe10**

La table 2.1 et les figures 2.1 et 2.2 illustrent la difference entre les deux ordonnanceur et l'apport de D'souza.

τ_i	C_i	T_i
1	1	10
2	4	23
3	3	36

TABLE 2.1 – Ensemble de taches

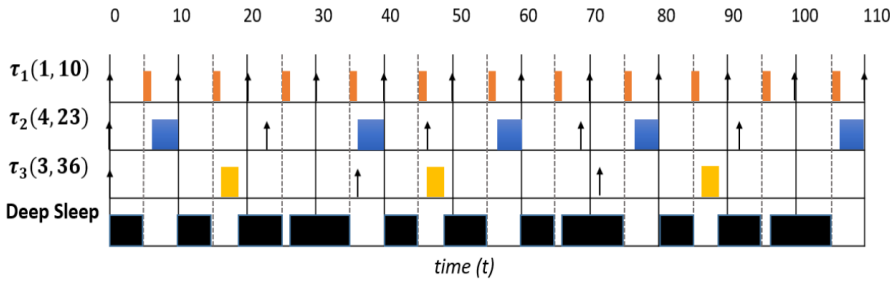

FIGURE 2.1 – Energy-Saving RHS

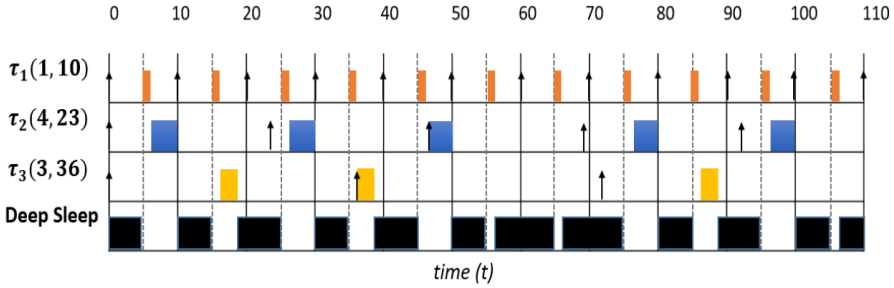

FIGURE 2.2 – Energy-Saving RHS+

2.6 Conclusion

Deuxième partie

Contributions

Chapitre 3

Endormissement de tache sous priorité Fixe

Sommaire

3.1 modéle de tâches

Le modèle utilisé ici est le modèle de tâche périodique de Liu et Layland défini au chapitre 1.

Soit $\mathcal{T}_{=}\{\tau_1, \tau_2, \cdots, \tau_n\}$ un ensemble de tache, chaque tache τ_i est caracterisé par $\tau_i = (\mathsf{C}_i, \mathsf{D}_i, \mathsf{T}_i)$ et :

— τ_i est periodique.

— C_i est le pire temps d'execution de la tache i.

— D_i est l'écheance relative de la tache i.

— T_i est la periode relative de la tache i.

— l'ensemble de tâches $\mathcal{T}$ est ordonnançable avec l'algorithme Deadline Monotonic.

3.2 Le cas monoprocesseur

Dans cette section nous allons inserer une tâche d'endormissement $\tau_{\text{SLEEP}} = \{\mathsf{C}_{\text{SLEEP}}, \mathsf{D}_{\text{SLEEP}}, \mathsf{T}_{\text{SLEEP}}\}$ avec $\min \mathsf{C}_{\text{SLEEP}} \leq \mathsf{C}_{\text{SLEEP}} \geq \max \mathsf{C}_{sleep}$ avec $\mathcal{C}^{sleepMax}_{=,(,1}(-)u_{)} \times \mathsf{T}_H$ et $\tau_{sleep} \cup \mathcal{T}$ est ordonnançable avec Deadline Monotonic.

Pour cela nous presentons l'algorithme d'insertion de tache d'endormissement τ_{SLEEP} dans un taskset $\mathcal{T}$.

Nous illustrons notre algorithme avec un exemple d'application. Le tableau 3.1 et la figure **??** represente un ensemble de tâches $\mathcal{T}$ ordonnançable par Deadline Monotonic où on a inseré une tache d'endormissement τ_{SLEEP} :

τ_i	C_i	D_i	T_i
1	1	10	10
2	2	15	15

TABLE 3.1 – Ensemble de tâches periodiques

3.3 Le cas multiprocesseur

Dans cette section nous allons inserer un ensemble de tâches d'endormissement $\{\tau^1_{\text{SLEEP}}, \tau^2_{\text{SLEEP}}, ..., \tau_{\text{S}}$ tel que $\tau^i_{\text{SLEEP}} = \{\mathsf{C}^i_{\text{SLEEP}}, \mathsf{D}^i_{\text{SLEEP}}, \mathsf{T}^i_{\text{SLEEP}}\}$ dans m processeurs $p = \{p_1, p_2, ..., p_m\}$. Pour cela nous presentons nous presentons deux strategie d'insertion :

Insertion Locale : Chaque processeur$_i$ à sa propre tache d'endormissement τ^i_{SLEEP}

Insertion Globale : Tous les processeur ont une même tache d'endormissement $\tau_{\text{SLEEP}} = \tau^1_{\text{SLEEP}} = \tau^2_{\text{SLEEP}} = ... = \tau^m_{\text{SLEEP}}$

Les deux algorithmes representent l'inseretion locale (Resp. globale) d'un ensemble de tâches d'endormissement dans un ensemble de processeur.

Nous illustrons notre algorithme avec un exemple d'application. Le tableau **??** et les figures **??** et **??** representent un ensemble de tâches $\mathcal{T}$ ordonnançable par Deadline Monotonic dans un multiprocesseur à 2 processeur en partitionnement FirstFit où on a inseré deux taches d'endormissements τ_{SLEEP} en locale et en globale.

τ_i	C_i	D_i	T_i
1	5	10	10
2	7	15	21
3	2	22	24

TABLE 3.2 – Ensemble de tâche périodique

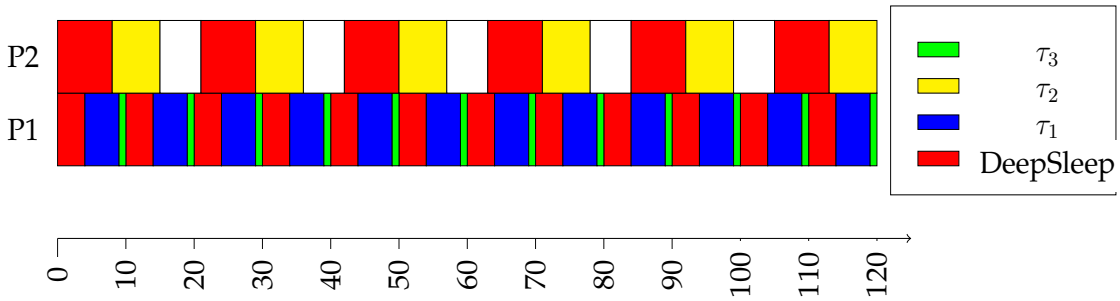

FIGURE 3.1 – Insertion locale de tâches d'endormissements dans un multi-processeur

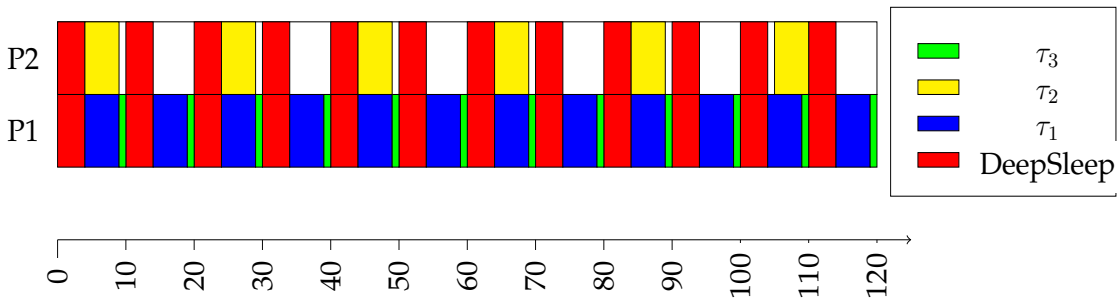

FIGURE 3.2 – Insertion globale de tâches d'endormissements dans un multi-processeur

Chapitre 4

Endormissement de tache sous priorité dynamique

Sommaire

4.1 modéle de tâches

Le modèle utilisé ici est le modèle de tâche sporadique.

Soit $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_n\}$ un ensemble de tache, chaque tache τ_i est caracterisé par $\tau_i = (\mathsf{C}_i, \mathsf{D}_i, \mathsf{T}_i)$ et :

— τ_i est sporadique.

— C_i est le pire temps d'execution de la tâche i

— D_i est l'écheance relative de la tâche i

— T_i est la periode de la tâche i

— l'ensemble de tâches $\mathcal{T}$ est ordonnançable avec l'algorithme EarliestDeadlineFirst

4.2 Limitation de nombre de preemption

4.3 Le cas monoprocesseur

τ_i	C_i	D_i	T_i
1	2	25	25
2	2	10	10

TABLE 4.1 – Ensemble de taches periodiques

4.4 Le cas multiprocesseur

τ_i	C_i	D_i	T_i
1	2	10	10
2	2	15	21

TABLE 4.2 – Insertion locale de tâches d'endormissements dans un multiprocesseur

4.5 insertion locale

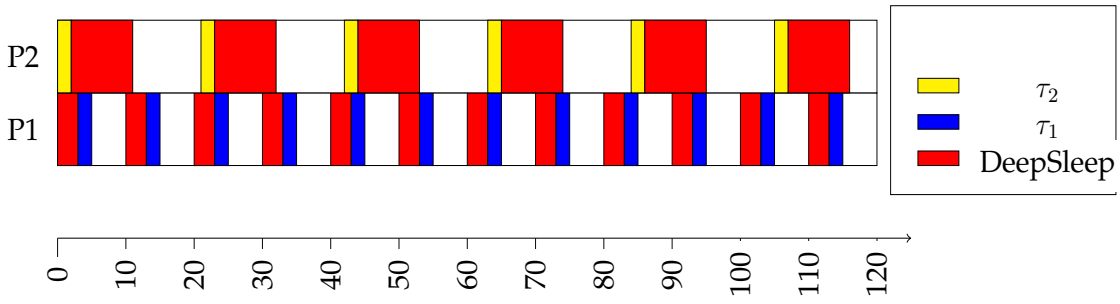

FIGURE 4.1 – Insertion globale de tâches d'endormissements dans un multiprocesseur

4.6 insertion globale

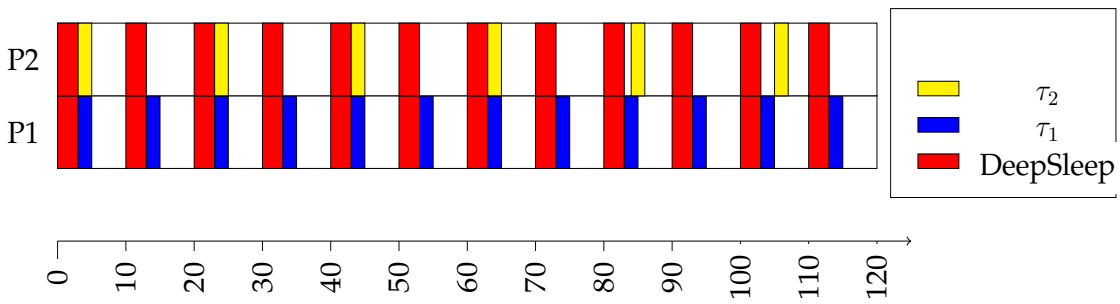

FIGURE 4.2

Chapitre 5

Expérimentations

5.1 La génération des taches

5.1.1 Cas monoprocesseur

Algorithmes UUnifast

L'algorithme UUnifast est un algorithme mis au point pour la génération de taux d'utilisation sur monoprocesseur. Il génère une distribution uniforme de n taux d'utilisation non biaisés à partir du nombre de tâches n de l'ensemble et du taux d'utilisation processeur total souhaité U. UUnifast est un algorithme efficace de complexité O(n). Nous rappelons qu'un ensemble au taux d'utilisation supérieur à 1 est trivialement non ordonnançable puisque l'utilisation processeur dépasse alors le temps maximal disponible.

Génération des périodes

Lors de la génération de tâches, le choix des périodes est un élément sensible pour les tests d'ordonnançabilité. En effet, certains de ces tests basent leur analyse sur un intervalle de faisabilité. La longueur de cet intervalle dépend du plus petit commun multiple des périodes (ppcm) appelée l'hyper-période. Si les périodes sont grandes, premières entre elles, l'hyperpériode explose. Le défi consiste donc à générer des périodes aléatoirement tout en limitant la taille de l'hyper-période et c'est l'objet de la méthode de Goossens et Macq **Goossens01**

Génération des échéances

Similairement à Goossens et Macq dans **Goossens01** nous generons les échéances des taches generées. Nous déterminons aléatoirement l'échéance dans un intervalle [$0.75 \times T_i$, T_i]. En résumé, $Di = \{T_i \times C_i) \times aleatoire(0.75 \times T_i, T_i)\} + C_i$ avec aleatoire(dmin, dmax) retourne un nombre réel pseudo-aléatoire uniformément distribué sur l'intervalle [dmin, dmax] et où la fonction arrondi(x) retourne l'entier le plus proche de x.

5.1.2 Cas multiprocesseur

Algorithme UUnifast-Discard

La méthode UUnifast présentée dans le cas monoprocesseur n'est pas utilisée en contexte multiprocesseur, lorsque le taux d'utilisation du processeur U peut dépasser 1. En effet, lorsque que le taux d'utilisation total dépasse 1, UUnifast présente le risque de générer des taux d'utilisation par tâche supérieurs à 1. Tâche qu'il n'est alors possible d'ordonnancer sur aucun processeur. Pour y remédier, Davis et Burns **DB11** ont proposé une extension appelée UUnifast-Discard. Elle consiste simplement à employer UUnifast avec U supérieur à 1 et à rejeter les ensembles pour lesquelles au moins un taux d'utilisation par tâche est supérieur à 1. Son implantation est simple mais cette méthode a l'inconvénient d'être particulièrement inefficace lorsque U approche $\frac{n}{2}$ **Emb10**

5.2 La simulation

5.3 Discussions