Table des matières

Table des figures

Liste des tableaux

Introduction Generale

Première partie

Partie Theorique

Chapitre 1

Introduction Au systeme temps réel

Sommaire

1.1 Introduction **10**

1.2 Taxonomie sur les systèmes temps réel **10**

1.3 Ordonnancement monoprocesseur **10**

1.3.1 Algorithme d'ordonnancement à priorité fixe 11

Rate Monotonic [?] 11

Deadline Monotonic [?] 12

1.3.2 Algorithme d'ordonnancement à priorité dynamique 13

Earliest Deadline First[?] 13

Least Laxity 14

1.4 Ordonnancement Multiprocesseur **15**

1.4.1 Classification 15

1.4.2 Optimalité 15

1.4.3 Algorithme d'ordonnancement utilisant une stratégie par partitionnement . 16

Généralité 16

First-Fit et Best-Fit 16

1.4.4 Algorithme d'ordonnancement utilisant une stratégie globale 16

Algorithme de type Pfair 16

1.5 Conclusion **16**

1.1 Introduction

1.2 Taxonomie sur les systèmes temps réel

Différents niveaux de criticité

Les systèmes temps réel dits critiques (ou dur) correspondent ont des systèmes pour lesquelles il est intolérable qu'une échéance soit manquée au 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 Ordonnancement monoprocesseur

Un algorithme d'ordonnancement est chargé de répartir les tâches sur un ou plusieurs processeurs : il décide quelle tâche sera exécutée sur tel processeur et pour combien de temps.

Definition

Nous définissons dans un premier temps les termes habituels concernant les systèmes 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, i.e. la préempter, 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.
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.

1.3.1 Algorithme d'ordonnancement à priorité fixe

Rate Monotonic [?]

Rate Monotonic est un algorithme à priorité fixe introduit par Liu et Layland dans [?]. Cet algorithme affecte des 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 ??. 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

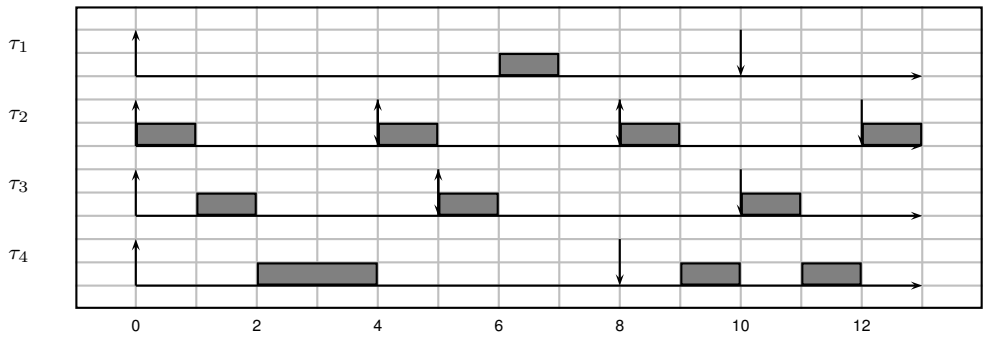

FIGURE 1.1 – Ordonnancement sous Rate Monotonic

Theoreme 1.3.1. *Rate Monotonic est optimal pour l'ordonnancement de systèmes de tâches synchrones, indépendantes et à échéance sur requête en présence de préemption.*

Theoreme 1.3.2 (Condition Suffisante [**?**])**.** *. Un système temps réel composé de n tâches est ordonnançable par Rate Monotonic si :*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{1.1}$$

Deadline Monotonic [?]

Deadline Monotonic est un algorithme à priorité fixe introduit par Leung et Whitehead dans [[**?**]]. 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.

Theoreme 1.3.3. *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) :

Theoreme 1.3.4. *Un système temps réel composé de n tâches est ordonnançable par Deadline Monotonic si la condition suivante est vérifiée :*

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

Condition necessaire et suffisante d'ordonnançabilité Joseph et al [**?**] 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 i de priorité p terminera son exécution quand les taches les plus prioritaire sont actifs avec elle en même temps.

Theoreme 1.3.5. *soit* $\Gamma = \tau_1, \tau_2, ...\tau_n$ *un ensemble de* n *taches.* Γ *est ordonnancable sous deadline monotonic ssi :*

$$\forall \tau_i \in \Gamma / R_i \leq D_i \tag{1.3}$$

$$R_i = \begin{cases} R_i^0 = C_i \\ R_i^{(k+1)} = C_i + \sum_{j \in pr(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil * C_j \end{cases} \tag{1.4}$$

τ_i	C_i	D_i	T_i	priorité	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

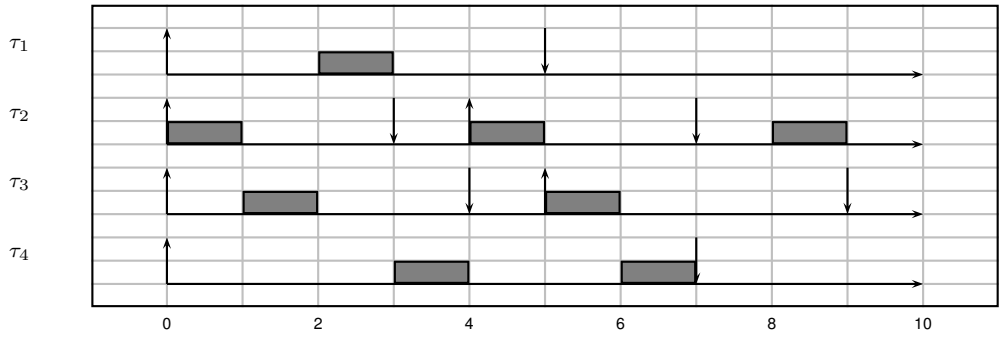

FIGURE 1.2 – Ordonnancement sous Deadline Monotonic

1.3.2 Algorithme d'ordonnancement à priorité dynamique

Les algorithmes à priorité dynamique affectent une priorité qui n'est 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.

Earliest Deadline First[?]

Earliest Deadline First est un algorithme connu et étudié depuis longtemps [?, ?, ?]. 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 8).

Theoreme 1.3.6 ([?]). *Un système de n tâches à échéance contrainte est ordonnançable par Earliest Deadline First si et seulement si :*

$$U \leq 1 \tag{1.5}$$

Condition necessaire et suffisante d'ordonnançabilité[?] il existe un test d'ordonnancabilité basée sur la fonction de la demande processeur DBF(Γ, t) causée par des tâches activées et devant être terminées dans l'intervalle [0, t].

$$DBF(\Gamma, t) = \sum_{\tau \in \Gamma} dbf(\tau, t) \tag{1.6}$$

$$dbf(\tau, t) = max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \times C_i\right) \tag{1.7}$$

Theoreme 1.3.7. *Un système Γ de n tâches à échéance contrainte est ordonnançable par Earliest Deadline First si et seulement si :*

$$\forall t \geq 0, DBF(\Gamma, t) \leq t \tag{1.8}$$

Theoreme 1.3.8 ([?]). *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

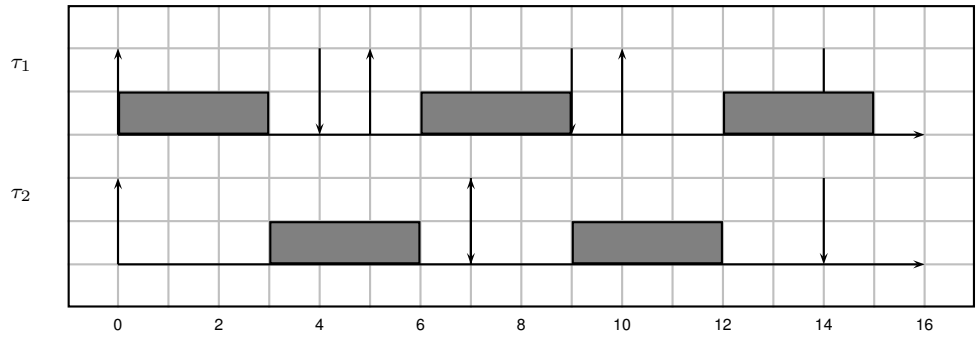

FIGURE 1.3 – Ordonnancement sous EDF

Least Laxity

L'algorithme Least Laxity [?]utilise la notion de laxité pour attribuer des priorités aux tâches.

Définition 9. La laxité (notée L) correspond à la longueur de l'intervalle de temps maximum pendant lequel la tâche peut ne pas avoir le processeur sans rater son échéance.

$$L_i = D_i - C_i \tag{1.9}$$

Par exemple, une tâche avec une laxité de 0 doit obligatoirement avoir le processeur jusqu'à sa terminaison sans quoi elle ratera son échéance et sera donc non ordonnançable. Le principe de l'algorithme est donc d'attribuer la plus haute priorité à l'instance dont la laxité est la plus faible, car c'est

l'instance ayant le moins de marge possible. Il est à noter que d'une part, cet algorithme nécessite une mise à jour des priorités des tâches à chaque instant, et mobilise dans ce but beaucoup de ressources de calcul, mais qu'en plus, il provoque de nombreux changements de contexte, également coûteux en temps.

Theoreme 1.3.9 ([?]). *Tout comme EDF, LL est optimal pour des systèmes de tâches indépendantes. Toutefois, au vue de ses inconvénients vis-à-vis du nombre de changements de contexte et du calcul des priorités nécessaire à chaque instant, LL est rarement utilisé en pratique.*

1.4 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 les problèmes suivants :

- 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.

Nous allons nous intéressé uniquement au problème de placement et d'ordonnancement des tâches, sans prendre en compte la migration et nous supposerons que les tâches sont indépendantes.

1.4.1 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 par partitionnement : 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.

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 non comparables [**?**].

1.4.2 Optimalité

Theoreme 1.4.1 ([?]). *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, alors il existe des algorithmes optimaux, comme les algorithmes de type Pfair par exemple.*

1.4.3 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 (Sac à dos), 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 [**?**]. 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, il existe deux heuristiques First Fit et Best Fit, qui reposent tous deux 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 :

- pour les algorithmes de type First Fit, la tâche est placée sur le premier processeur avec un ensemble de tache ordonnançable ;
- pour les algorithmes de type Best Fit, la tâche est placée sur le processeur avec un ensemble de tache ordonnançable et ayant le taux d'utilisation maximal.

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

1.4.4 Algorithme d'ordonnancement utilisant une stratégie globale

Les algorithmes d'ordonnancement utilisant une stratégie globale n'entrant pas dans le cadre de ce mémoire, nous ne présenterons que succinctement les algorithmes les plus utilisés.

Algorithme de type Pfair

Les algorithmes de type Pfair ont la particularité d'exécuter les tâches à un taux régulier. Un algorithme Pfair a une caractéristique de limiter les variations du taux d'utilisation en s'assurant que le taux d'exécution de la tâche T_i reste voisin de U_i, quelle que soit la longueur de l'intervalle considéré.

Parmi les algorithmes d'ordonnancement Pfair, nous pouvons citer :

- PF [**?**]
- PD [**?**]
- PD2 [**?**]

1.5 Conclusion

Chapitre 2

Etat de l'art

Sommaire

2.1 Introduction **17**

2.2 Modélisation des tâches **17**

2.3 Les modèles de consommation d'énergie DVFS et DPM **19**

2.3.1 Consommation dynamique 19
2.3.2 Consommation statique 20

2.4 Les états C-states du processeur **20**

2.5 L'endormissement de processeur (Online VS Offline) **22**

2.6 Le Modèle d'endormissement de Dsouza **22**

2.6.1 Algorithme d'ordonnancement « Energy-Saving Rate-Harmonized Scheduling » 23
2.6.2 Energy-Saving Rate-Harmonized Scheduling 23
2.6.3 Energy-Saving Rate-Harmonized Scheduling+ 24

2.7 Conclusion **25**

2.1 Introduction

2.2 Modélisation des tâches

Notre modélisation d'un système temps réel est basée sur celle proposée par Liu et Layland [**?**]. Soit un système temps réel composé d'un ensemble de tâches nommé Γ qui comprend n tâches pério-

diques dont les caractéristiques sont détaillées ci-dessous. Nous définissons dans cette section tous les termes qui seront utilisés en relation avec la notion d'ensemble de tâches.
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. Le modèle de tâches que nous utilisons est le modèle de tâches dit périodique pour l'ordonnancement fixe et sporadique pour l'ordonnancement dynamique.
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 τi libère ses travaux périodiquement suivant sa période Ti, 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 de l'ensemble de tâches correspond au plus petit commun multiple de toutes les périodes de l'ensemble de tâches. $H = PPCM(T0,T_1, ..., T_n)$ Le nombre de tâches dans un système temps réel embarqué est limité et les périodes de ces tâches ont en général des relations temporelles entre elles. Par exemple, il est peu probable que les périodes des tâches soient premières entre elles, les périodes des tâches sont souvent des harmoniques. En prenant un exemple concret, des tâches peuvent avoir des périodes de 1ms, 2ms, 5ms ou 10ms mais il est moins fréquent de trouver des tâches avec des périodes de 1.78ms et de 8.54ms. La valeur de l'hyperpériode ainsi que le nombre de travaux dans une hyperpériode restent donc naturellement raisonnables.
Date Réveil : notée O_i, c'est la date où la tache libère son premier travail, chaque travail de la tâche est libéré à l'instant $O_i + kT_i$ avec $K \in \mathbb{N}$
Pire temps d'execution (Worst Case Execution Time WCET) : est la durée maximale de l'exécution de chacun de ses travaux. Le WCET de la tâche τi est noté Ci. 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 [**?**] pour plus d'informations. Nous supposons que le WCET de chaque travail est connu.
Échéance (Deadline) : Chaque travail une fois libéré doit terminer son exécution avec une certaine date sous peine de violer son échéance. Nous notons Di l'échéance relative de la tâche τi. L'échéance absolue j.d du travail j sera donc la date de sa libération additionnée de cette échéance relative.

Il existe trois types de modèles de taches :

- Modèles à « échéances implicites » où l'échéance de chaque travail égale à sa période $T_i = D_i$.
- Modèles à « échéances contraintes » où l'échéance de chaque travail est inférieure ou égale à sa période $D_i <= T_i$.
- 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 WCET et sa période. L'utilisation U_i de la tâche τ_i est donc $\frac{C_i}{T_i}$
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 :

$$U = \sum_{i=1}^{n} U_i \tag{2.1}$$

Toutes les notations relatives à l'ensemble de tâches sont résumées dans le Tableau **??**. D'autres notations seront introduites par la suite.

Γ	Ensemble de taches
H	Hyperperiode
τ_i	Tache
C_i	WCET de la tache i
D_i	Echeance de la tache i
T_i	Periode de la tache i
U_i	Utilisation de la tache i

TABLE 2.1 – Notations

2.3 Les modèles de consommation d'énergie DVFS et DPM

L'énergie consommée par un processeur est le produit 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.

2.3.1 Consommation dynamique

La fréquence et la tension d'alimentation des processeurs sont liées. Lorsqu'un processeur possède plusieurs fréquences de fonctionnement, chaque fréquence peut fonctionner avec au maximum seulement une ou deux tensions d'alimentation et la consommation énergétique du processeur dépend à la fois de la fréquence et de la tension d'alimentation utilisées.

La puissance dynamique est dissipée lors de la commutation des composants et dépend donc de la fréquence f du processeur. Elle peut être approximée selon la relation suivante [?] :

$$P_{dynamique} = C * f * V_{dd}^2 \tag{2.2}$$

Où C correspond à la capacité de sortie du circuit et V_{dd} à la tension d'alimentation.
Les solutions permettant de réduire la consommation 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 [?, ?, ?].

2.3.2 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 [?, ?]. 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 [?] affirme que l'énergie statique représente jusqu'à 70% de la consommation énergétique totale dissipée par un processeur. Plusieurs études confirment cette affirmation [?, ?, ?, ?, ?, ?]. Des expériences ont également été faites [?, ?, ?] 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.

2.4 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 :

$$Pen_s = \frac{1}{2} \times Del_s \times (Cons_{actif} - Cons_s) \tag{2.3}$$

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. [**?**, **?**]). 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 [**?**, **?**, **?**]. Chaque état basse-consommation possède donc son propre BET et nous nommons BET_s 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 [**?**] utilisé par Awan et al. [**?**, **?**, **?**]. Ce processeur est basé sur une architecture PowerPC. Ses états basse-consommation sont détaillés dans le Tableau 2.4.

2.5 L'endormissement de processeur (Online VS Offline)

2.6 Le Modèle d'endormissement de Dsouza

Anthony Rowe a proposé un algorithme d'ordonnancement appellé "Energy Saving Rate Harmonized Scheduling (ES RHS)" [**?**] 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".

subsectionPériode d'harmonisation

Soit un ensemble Γ de tache de n taches périodiques indépendantes, l'ensemble de tâche est ordonnée selon les périodes de chaque tache de tel façon que $T_1 \leq T_2 \leq \cdots \leq T_n$. Un ensemble de taches $\{\tau_1, \tau_2, \ldots, \tau_n\}$ est harmonique si il existe un T_H tel que $\forall \tau_i / \frac{T_i}{T_h} \in \mathbb{Z}$

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

En général RHS utilise un ensemble de Valeurs périodiques $\{ T_H^1, T_H^2, \ldots, T_H^3 \}$ où $T_H^i \leq T_i$ pour i = 1 .. 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 $\tau_1 = O_1 = 0$.
Les tâches $\{\tau_1, \tau_2, \ldots, \tau_n\}$ 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 [**?**].
Dans le L'ordonnancement RHS de base, $\{ T_H^1 = T_H^2 = ... = T_H^n = T_H \}$, T_H Est simplement appelé période d'harmonisation et $T_H \leq T_1$. 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 [**?**]. Supposer $\Psi = \{ \tau_j \mid T_j T_1, j \neq 1 \}$. si $\Psi = \emptyset, T_H = T_1$. sinon $T_H = \frac{T_1}{2}$.

Theoreme 2.6.1 ([**?**]). *un ensmeble de taches Γ est faisable sous RHS si $U \leq 0.5$.*

Theoreme 2.6.2. *le calcul du pire temps de reponse W_k [23,24] dans le cas de RHS peut etre redéfini comme suit :*

$$\begin{aligned} W_0 &= C_i + T_H \\ W_{k+1} &= C_i + T_H + \sum_{j=1}^{i-1} \left\lceil \frac{W_k}{T_j} \right\rceil C_j \end{aligned} \tag{2.4}$$

Theoreme 2.6.3 (citeRowe10). *Soit $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ un ensemble de tache periodiques indépendantes.*
Si il existe une tache τ_i avec un pire temps de reponse W_k tel que $W_k > D_i$ alors l'ensemble de tache Γ est non ordonnançable.

2.6.2 Energy-Saving Rate-Harmonized Scheduling

comme vu dans la section **??**, 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 $PW_{active} >_{idle} \gg_{sleep}$ et $_{idle} \ll_{sleep}$.
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 $_{sleep}$ 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 [?] a redéfini la notion d'harmonisation pour ameliorer l'ordonnancemen ES-RHS.

2.6.3 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 [?].
la table ?? et les figures ?? et ?? 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 2.2 – Ensemble de taches

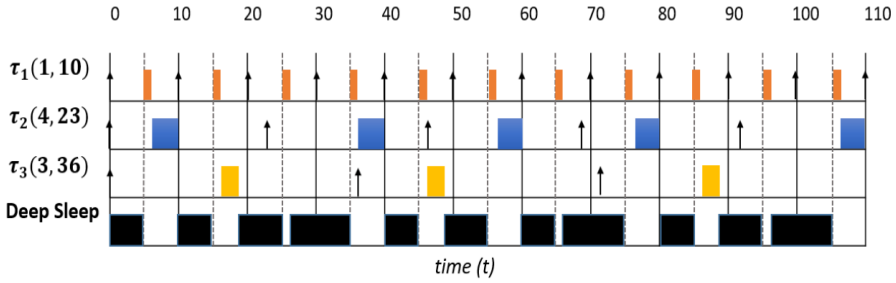

FIGURE 2.1 – Energy-Saving RHS

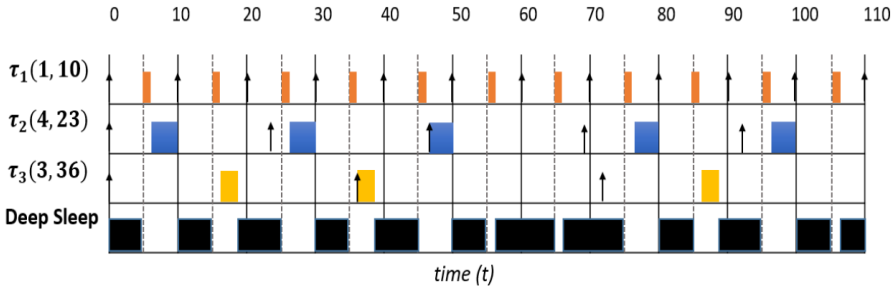

FIGURE 2.2 – Energy-Saving RHS+

2.7 Conclusion

Deuxième partie

Contributions et Implementation

Chapitre 3

Endormissement de tache sous priorité Fixe

Sommaire

3.1 Le cas monoprocesseur **29**

3.2 Le cas multiprocesseur **29**

3.1 Le cas monoprocesseur

3.2 Le cas multiprocesseur

Chapitre 4

Endormissement de tache sous priorité dynamique

Sommaire

4.1 Le cas monoprocesseur **31**

4.2 Le cas multiprocesseur **31**

4.1 Le cas monoprocesseur

4.2 Le cas multiprocesseur

Chapitre 5

Expérimentations

5.1 La génération des taches

5.2 La simulation

5.3 Discussions

Conclusion Generale

Bibliographie

[1] MPC5510 Microcontroller Family Reference Manual.

[2] RM0038 Reference manual STM32l100xx, STM32l151xx, STM32l152xx and STM32l162xx advanced ARM®-based 32-bit MCUs.

[3] James H. Anderson and Anand Srinivasan. Early-release fair scheduling. In Proceedings of the 12th Euromicro Conference on Real-time Systems, Euromicro-RTS'00, pages 35–43, Washington, DC, USA, 2000. IEEE Computer Society.

[4] T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabarti, and W. Wolf. Mobile supercomputers. Computer, 37(5) :81–83, May 2004.

[5] Muhammad Ali Awan and Stefan M Petters. Enhanced race-to-halt : A leakage-aware energy management approach for dynamic priority systems. In 23rd Euromicro Conference on Real-Time Systems (ECRTS), 2011, pages 92–101. IEEE, 2011.

[6] Muhammad Ali Awan and Stefan M Petters. Energy-aware partitioning of tasks onto a heterogeneous multi-core platform. In Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th, pages 205–214. IEEE, 2013.

[7] Muhammad Ali Awan, Patrick Meumeu Yomsi, and Stefan M. Petters. Optimal procrastination interval for constrained deadline sporadic tasks upon uniprocessors. In Proceedings of the 21st International Conference on Real-Time Networks and Systems, RTNS '13, pages 129–138, New York, NY, USA, 2013. ACM.

[8] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress : A notion of fairness in resource allocation. Algorithmica, 15(6) :600–625, 1996.

[9] Sanjoy K Baruah, Johannes E Gehrke, and C Greg Plaxton. Fast scheduling of periodic tasks on multiple resources. In Parallel Processing Symposium, 1995. Proceedings., 9th International, pages 280–288. IEEE, 1995.

[10] Sanjoy K. Baruah, Rodney R. Howell, and Louis E. Rosier. Feasibility problems for recurring tasks on one processor. Theor. Comput. Sci., 118(1) :3–20, 1993.

[11] S. Borkar. Design challenges of technology scaling. IEEE Micro, 19(4) :23–29, Jul 1999.

[12] J. J. Chen and C. F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), pages 28–38, Aug 2007.

[13] Hui Cheng and S. Goddard. Online energy-aware i/o device scheduling for hard real-time systems. In Proceedings of the Design Automation Test in Europe Conference, volume 1, pages 6 pp.–, March 2006.

[14] Michael L. Dertouzos. Control robotics : The procedural control of physical processes. In IFIP Congress, pages 807–813, 1974.

[15] Vinay Devadas and Hakan Aydin. On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications. In Proceedings of the 8th ACM International Conference on Embedded Software, EMSOFT '08, pages 99–108, New York, NY, USA, 2008. ACM.

[16] Kwang Soo Hong and Joseph Y.-T. Leung. On-line scheduling of real-time tasks. IEEE Trans. Computers, 41(10) :1326–1331, 1992.

[17] W. A. Horn. Some simple scheduling algorithms. Naval Research Logistics Quarterly, 21(1) :177–185, 1974.

[18] Hongtao Huang, Feng Xia, Jijie Wang, Siyu Lei, and Guowei Wu. Leakage-aware reallocation for periodic real-time tasks on multicore processors. CoRR, abs/1011.3087, 2010.

[19] Kai Huang, Luca Santinelli, Jian-Jia Chen, Lothar Thiele, and Giorgio C. Buttazzo. Applying real-time interface and calculus for dynamic power management in hard real-time systems. Real-Time Systems, 47(2) :163–193, 2011.

[20] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. Comput. J., 29(5) :390–395, 1986.

[21] Nam Sung Kim, Todd M. Austin, David Blaauw, Trevor N. Mudge, Krisztiﾮn Flautner, Jie S. Hu, Mary Jane Irwin, Mahmut T. Kandemir, and Narayanan Vijaykrishnan. Leakage current : Moore's law meets static power. IEEE Computer, 36(12) :68–75, 2003.

[22] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling : The laws of diminishing returns. In Proceedings of the 2010 International Conference on Power Aware Computing and Systems, HotPower'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[23] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. Perform. Eval., 2(4) :237–250, 1982.

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

[25] Aloysius Ka-Lau Mok. Fundamental Design Problems of Distributed Systems for the Hard–Real–Time Environment. Technical Report MIT-LCS-TR-297, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1983. Ph.D. Thesis.

[26] Padmanabhan Pillai and Kang G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In Keith Marzullo and M. Satyanarayanan, editors, SOSP, pages 89–102. ACM, 2001. Operating System Review 35(5).

[27] Euiseong Seo, Jinkyu Jeong, Seonyeong Park, and Joonwon Lee. Energy efficient scheduling of real-time tasks on multicore processors. IEEE Trans. Parallel Distrib. Syst., 19(11) :1540–1552, 2008.

[28] Tajana Simunic, Luca Benini, Andrea Acquaviva, Peter Glynn, and Giovanni De Micheli. Dynamic voltage scaling and power management for portable systems. In DAC '01 : Proceedings of the 38th conference on Design automation, pages 524–529, New York, NY, USA, 2001. ACM.

[29] David Snowdon, Sergio Ruocco, and Gernot Heiser. Power management and dynamic voltage scaling : Myths and facts.

[30] David C. Snowdon, Stefan M. Petters, and Gernot Heiser. Power measurement as the basis for power management. In IN 2005 WS OPERAT. SYSTEM PLATFORMS FOR EMBEDDED REAL-TIME APPLICATIONS, 2005.

[31] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In OSDI '94 : Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[32] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenstrom. The worst-case execution time problem overview of methods and survey of tools. ACM Trans. Embedded Comput. Syst., 7(3), 2008. http ://dblp.uni-trier.de/db/journals/tecs/tecs7.htmlWilhelmEEHTWBFHMMPPSS08.

[33] F. Frances Yao, Alan J. Demers, and Scott Shenker. A scheduling model for reduced cpu energy. In FOCS, pages 374–382. IEEE Computer Society, 1995.