Ordonnancement temps réel multiprocesseur pour la réduction de la consommation énergétique des systèmes embarqués

Vincent Legout

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2014-ENST-0019

EDITE - ED 130

École doctorale n°130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « Informatique et Réseaux »

présentée et soutenue publiquement par

Vincent LEGOUT

08 Avril 2014

Ordonnancement temps réel multiprocesseur pour la réduction de la consommation énergétique des systèmes embarqués

Directeur de thèse : **Laurent PAUTET**
Co-encadrement de la thèse : **Mathieu JAN**

Jury	
Alix MUNIER, LIP6	Présidente
Laurent GEORGE, UPEM	Rapporteur
Yvon TRINQUET, IRCCYN	Rapporteur
Michel AUGUIN, UNICE	Examinateur
Isabelle PUAUT, IRISA	Examinateur
Mathieu JAN, CEA LIST	Co-Directeur de thèse
Laurent PAUTET, Télécom ParisTech	Directeur de thèse

TELECOM ParisTech
école de l'Institut Mines-Télécom - membre de ParisTech
46 rue Barrault 75013 Paris - (+33) 1 45 81 77 77 - www.telecom-paristech.fr

THÈSE

Remerciements

Je voudrais remercier en premier lieu Mathieu Jan et Laurent Pautet pour leur encadrement durant ces trois années. Merci pour toutes leurs critiques toujours pertinentes, leur patience ainsi que pour les nombreuses relectures, spécialement lors de la rédaction de ce manuscrit. C'est un plaisir d'avoir pu travailler avec eux. Je pense à notre premier entretien fin 2010 et je me dis avoir évolué et avoir appris au cours de ces trois années, merci.

Je voudrais ensuite remercier Laurent George et Yvon Trinquet d'avoir accepté de rapporter cette thèse. Merci également aux examinateurs Alix Munier, Isabelle Puaut et Michel Augin.

Merci au CEA de m'avoir permis de travailler dans de bonnes conditions et de m'avoir permis d'assister à des conférences en France et à l'étranger durant ces trois ans. Merci à toutes les personnes que j'ai cotoyées tout au long de cette thèse, notamment les membres du LaSTRE. Je garde un très bon souvenir de ces trois ans, que ce soit dans le cadre du CEA ou à l'extérieur, au batiment 451 ou à NanoInnov.

Merci particulièrement à Matthieu de m'avoir accepté en stage ainsi que pour son accompagnement qui m'a motivé pour postuler pour une thèse. Je voudrais également remercier Frédéric pour les discussions toujours enthousiastes et Thomas pour son aide et nos discussions. Merci également à Oana et Andreï ainsi qu'à Kods et Stéphane de m'avoir hebergé ces derniers jours. Un très grand merci à tout ceux qui ont relu cette thèse, je repense particulièrement à Julien et Lilia. Merci à tous les doctorants passés et actuels du LaSTRE. Et par ordre alphabétique, merci à Cyril, Emmanuel, François, Hela, Hugo, Ilias, José, Karl, Loïc, Moez, Nicolas, Pascal, Safae, Sergiu, Simon, Stéphane et tout ceux qui ne sont pas nommés ici.

Sur un point personnel, merci pour leur immense soutien à mes parents Odile et André qui m'hébergent pour écrire ses lignes, ainsi qu'à ma soeur et mon frère, Anne et Bertrand.

En espérant ne jamais oublier N.

Car j'ignore où tu fuis, tu ne sais où je vais,
Ô toi que j'eusse aimée, ô toi qui le savait !
A une passante, Charles Baudelaire

Résumé

Les systèmes temps réel sont souvent utilisés dans un contexte embarqué. Réduire leur consommation énergétique est par conséquent devenu un enjeu important, notamment pour augmenter leur autonomie. Nous nous intéressons plus précisément dans le cadre de cette thèse aux systèmes multiprocesseurs qui tendent à remplacer les systèmes monoprocesseurs dans le domaine du temps réel embarqué.

La consommation énergétique des systèmes embarqués temps réel est majoritairement due aux processeurs. Cette consommation énergétique est divisée en énergie statique et énergie dynamique. Les avancées technologiques ont entrainé une augmentation de la consommation statique au détriment de la consommation dynamique. Les algorithmes d'ordonnancement existants se focalisant essentiellement sur la réduction de la consommation dynamique, de nouvelles solutions sont nécessaires pour réduire la consommation statique en exploitant les états basse-consommation des processeurs. Dans un état basse-consommation, les consommations dynamique et statique sont fortement réduites mais un délai de transition et une pénalité sont nécessaires pour revenir à l'état actif.

Nous proposons dans cette thèse des algorithmes d'ordonnancement temps réel multiprocesseurs optimaux pour des systèmes temps réel dur et des systèmes temps réel à criticité mixte. Ce sont à notre connaissance les premiers algorithmes d'ordonnancement multiprocesseurs optimaux dont l'objectif est de réduire la consommation énergétique. Ces algorithmes d'ordonnancement permettent d'exécuter les tâches du système de telle sorte que la taille des périodes d'inactivité soit optimisée pour activer les états basse-consommation les plus économes en énergie. Chaque algorithme d'ordonnancement est divisé en deux parties. La première partie hors-ligne génère un ordonnancement en utilisant la programmation linéaire en nombres entiers pour minimiser la consommation énergétique. La seconde partie est en-ligne et augmente la taille des périodes d'inactivité lorsque les tâches terminent leur exécution plus tôt que prévu. Dans le cadre des systèmes temps réel à criticité mixte, nous profitons du fait que les tâches de plus faible criticité peuvent tolérer des dépassements d'échéances pour être plus agressif hors-ligne afin de réduire davantage la consommation énergétique.

Ces solutions sont ensuite évaluées puis comparées aux meilleurs algorithmes d'ordonnancement multiprocesseurs optimaux à l'aide d'un simulateur. Les résultats montrent que les algorithmes proposés utilisent de manière plus efficace les états basse-consommation. La consommation énergétique lorsque ceux-ci sont activés est en effet jusqu'à dix fois plus faible qu'avec les algorithmes d'ordonnancement multiprocesseurs existants.

Abstract

Real-time systems are often used in embedded context. Thus reducing their energy consumption is a growing concern to increase their autonomy. In this thesis, we target multiprocessor embedded systems which are becoming standard and are replacing uniprocessor systems.

We aim to reduce the energy consumption of the processors which are the main components of the system. It includes both static and dynamic consumption and it is now dominated by static consumption as the semiconductor technology moves to deep sub-micron scale. Existing solutions mainly focused on dynamic consumption. On the other hand, we target static consumption by efficiently using the low-power states of the processors. In a low-power state, the processor is not active and the deeper the low-power state is, the lower is the energy consumption but the higher is the transition delay to come back to the active state.

In this thesis, we propose new optimal multiprocessor real-time scheduling algorithms. They optimize the duration of the idle periods to activate the most appropriate low-power states. We target hard real-time systems with periodic tasks and also mixed-criticality systems where tasks with lower criticalities can tolerate deadline misses, therefore allowing us to be more aggressive while trying to reduce the energy consumption. Offline, we use an additional task to model the idle time and we use mixed integer linear programming to compute a schedule minimizing the energy consumption. Online, we extend an existing scheduling algorithm to increase the length of the existing idle periods. To the best of our knowledge, these are the first optimal multiprocessor scheduling algorithms minimizing the static energy consumption.

Evaluations have been performed using existing optimal multiprocessor real-time scheduling algorithms. Results show that the energy consumption while processors are idle is up to ten times reduced with our solutions compared to the existing multiprocessor real-time scheduling algorithms.

Table des matières

CHAPITRE 1

Introduction

Sommaire

1.1 Motivations

Un système temps réel est un système avec des contraintes temporelles fortes [Sta88, Fis07] : la validité du système ne dépend pas seulement du résultat mais également de la date à laquelle le résultat a été obtenu. Ces systèmes sont de plus en plus présents autour de nous, que ce soit dans les produits grand public ou industriels. Leur étude est un domaine actif de la recherche, notamment pour exploiter au mieux les capacités matérielles tout en respectant les contraintes temporelles. Les travaux existants ont notamment cherché à améliorer l'ordonnançabilité de ces systèmes, c'est-à-dire garantir qu'un nombre maximum de tâches puissent être exécutées sur le système tout en garantissant qu'aucune échéance ne soit violée.

Un système embarqué est un système soumis à certaines contraintes en raison de son environnement. Contrairement à un système classique, il est situé dans un environnement plus ou moins « hostile » avec principalement des contraintes d'énergie et d'espace. Par exemple, les constructeurs automobiles intègrent maintenant de plus en plus de systèmes embarqués dans leurs véhicules et ces systèmes doivent prendre le moins de place possible pour s'intégrer dans l'habitacle du véhicule. Ce travail ne traite que des systèmes embarqués et nous ne préciserons plus systématiquement par la suite le caractère embarqué de ces systèmes. Notons néanmoins qu'un système embarqué peut être temps réel ou non.

Dans le cadre de cette thèse, nous nous intéressons aux systèmes multiprocesseurs, c'est-à-dire des systèmes avec plusieurs cœurs de calcul. En effet, les avancées technologiques rendant maintenant difficile l'augmentation de la fréquence de chaque processeur, l'accroissement des capacités de calcul des systèmes se fait en augmentant le nombre de processeurs. Les systèmes multiprocesseurs sont donc de plus en plus présents même dans le cadre des systèmes temps réel embarqués. Les systèmes monoprocesseurs vont ainsi être remplacés par des systèmes multiprocesseurs. Les industriels seront obligés de se tourner vers des systèmes multiprocesseurs. Toutes ces raisons justifient l'étude des systèmes multiprocesseurs. Dans le cadre de systèmes temps réel multiprocesseurs embarqués, le nombre de processeurs est classiquement compris entre 2 et 4 processeurs.

Les systèmes disposant maintenant de capacités de calcul plus importantes, il devient nécessaire pour exploiter ces possibilités de pouvoir exécuter sur le même système des tâches avec différents niveaux de criticité, c'est-à-dire des tâches critiques mais également des tâches dont la criticité est moindre comme par exemple des tâches de supervision ou de contrôle. Ces systèmes sont généralement regroupés sous la définition de systèmes temps réel à criticité mixte. En plus des systèmes temps réel dur, nous souhaitons nous intéresser à ces systèmes qui vont devenir de plus en plus présents et pour lesquels aucune solution n'a encore été proposée afin de réduire la consommation énergétique.

La motivation principale pour réduire la consommation énergétique des systèmes embarqués temps réel multiprocesseurs est d'augmenter leur autonomie car ces systèmes embarqués sont souvent dans des environnements hostiles où la source d'énergie principale a une capacité réduite (e.g. une batterie). Réduire leur consommation énergétique permet également de réduire l'encombrement car les systèmes ayant une consommation énergétique réduite nécessitent de plus petites sources d'énergies et moins de mécanismes de refroidissement [Gru02].

Plus généralement, réduire notre consommation énergétique est également aujourd'hui un objectif sociétal non spécifique aux systèmes temps réel. Les ressources énergétiques fossiles se raréfiant et la consommation énergétique mondiale augmentant rapidement, il paraît important de réduire la consommation énergétique de tous les systèmes du fait de leur prolifération et de leur importance dans notre société. Une consommation énergétique réduite est également maintenant devenue un argument de vente important.

La consommation énergétique des systèmes temps réel est constituée des consommations dynamique et statique, qui sont dues respectivement à l'activité des transistors du circuit et aux courants de fuite [AP11]. Dans le cadre de cette thèse, nous nous intéressons uniquement à la consommation énergétique des processeurs, qui sont les principaux composants et qui sont responsables de la majorité de la consommation énergétique [SBDM99, CH10]. Les autres composants du système comme la mémoire ou les périphériques consomment également de l'énergie mais leur activité est liée à celle du processeur qui est responsable de leur activation. La consommation statique ayant évoluée et étant maintenant responsable de la majorité de la consommation énergétique [KAB+03, HSC+11], il est nécessaire d'utiliser les états basse-consommation des processeurs, seule solution pour réduire la consommation énergétique. Un délai de transition et une pénalité énergétique sont par contre nécessaires pour revenir d'un état basse-consommation à l'état actif.

Cette thèse est donc motivée par le constat que les processeurs disponibles aujourd'hui

ont évolués par rapport à ceux disponibles lorsque les première études sur la réduction de la consommation énergétique ont été publiées (e.g. [YDS95]). La modélisation de la consommation énergétique utilisée alors n'est plus adaptée aux processeurs actuels, les composants responsables de la majorité de la consommation énergétique ayant évolués. Il est donc nécessaire de revoir cette modélisation, tout comme les algorithmes d'ordonnancement temps réel dont l'objectif est de réduire la consommation énergétique. L'étude de l'ordonnancement des systèmes temps réel dans le but de réduire leur consommation énergétique a en effet commencé depuis quelques dizaines d'années et des solutions efficaces existent pour réduire la consommation énergétique des systèmes temps réel monoprocesseurs. Mais l'arrivée de systèmes multiprocesseurs rend nécessaire une remise à niveau de ces travaux pour pouvoir exploiter au mieux les nouvelles possibilités offertes. Les algorithmes d'ordonnancement temps réel multiprocesseurs ne peuvent en effet être considérés comme une mise à l'échelle des algorithmes monoprocesseurs [Ber07]. Les solutions pour les systèmes monoprocesseurs ne peuvent donc pas être adaptées pour les systèmes multiprocesseurs et de nouveaux algorithmes d'ordonnancement doivent par conséquent être proposés.

1.2 Objectif

L'objectif de cette thèse est de proposer des algorithmes d'ordonnancement multiprocesseur pour réduire la consommation énergétique. Contrairement aux solutions actuelles qui se concentrent sur la réduction de la consommation dynamique, notre objectif est de réduire à la fois la consommation dynamique et la consommation statique en utilisant les états basse-consommation des processeurs. Dans ces états basse-consommation, la consommation dynamique est nulle et la consommation statique fortement réduite suivant l'efficacité de l'état basse-consommation. Pour atteindre cet objectif, l'algorithme d'ordonnancement doit optimiser la taille des périodes d'inactivité pour permettre d'utiliser les états basse-consommation les plus efficaces d'un point de vue énergétique. Optimiser la taille des périodes d'inactivité permet également de minimiser le nombre de transitions et les pénalités associées pour revenir à l'état actif.

Les solutions actuelles dont l'objectif est de réduire la consommation énergétique n'ont pas de point de vue global sur tout l'ordonnancement, elles prennent leurs décisions d'ordonnancement lors de l'exécution et se concentrent donc sur l'extension de la taille de la période d'inactivité courante pour que l'état basse-consommation le plus efficace puisse être activé. Notre objectif au contraire est de ne pas se concentrer sur une seule période d'inactivité mais sur l'ensemble des périodes d'inactivité afin d'optimiser l'utilisation des états basse-consommation durant toute l'exécution du système et ainsi réduire la consommation statique.

Nous souhaitons également réduire la consommation énergétique lors de l'exécution lorsque les tâches temps réel ne consomment pas leur pire temps d'exécution et libèrent du temps processeur. Pour les systèmes temps réel dur, il est nécessaire d'être pessimiste pour garantir toutes les échéances des tâches, il est donc essentiel d'attendre que chaque tâche ait fini son exécution pour utiliser ce temps processeur libéré. Dans le cadre des systèmes temps réel à criticité mixte où certaines tâches peuvent autoriser des violations d'échéances, il est

possible d'être plus optimiste et notre objectif est alors d'être plus agressif pour réduire la consommation énergétique tout en minimisant le risque de violations d'échéances.

1.3 Contributions

Pour atteindre les objectifs mentionnés précédemment, nous proposons plusieurs contributions sous la forme d'algorithmes d'ordonnancement multiprocesseurs temps réel optimaux pour réduire la consommation énergétique des systèmes temps réel embarqués en activant les états basse-consommation des processeurs. Ces algorithmes sont basés sur une même approche mais avec des objectifs et des implémentations différentes.

Notre première contribution est une approche permettant de construire un algorithme d'ordonnancement multiprocesseur optimal pour réduire la consommation énergétique. Cette approche hors-ligne permet d'avoir une vision globale de l'ordonnancement sur une hyperpériode et donc de pouvoir optimiser la taille des périodes d'inactivité pour activer les états basse-consommation les plus efficaces. Pour modéliser le temps d'inactivité du système et les périodes d'inactivité, cette approche ajoute une tâche temps réel supplémentaire à l'ensemble de tâches. L'ordonnancement hors-ligne est construit en utilisant la programmation linéaire en nombres entiers. L'objectif est de minimiser la consommation statique et les contraintes sont les respects des échéances. Cette première contribution a fait l'objet d'une publication au *First Workshop on Highly-Reliable Power-Efficient Embedded Designs (HARSH)* [LJP13b] et à l'École d'Été Temps Réel (ETR'13) [LJP13c].

Cette contribution a ensuite été étendue dans sa partie hors-ligne pour générer un ordonnancement hors-ligne adapté aux caractéristiques de chaque état basse-consommation. Pour améliorer l'efficacité énergétique lors de l'exécution, un algorithme d'ordonnancement en-ligne a également été proposé pour exploiter la différence entre les temps d'exécution des tâches attendus et ceux observés en pratique. Cet algorithme réduit la consommation énergétique en augmentant la taille des périodes d'inactivité existantes. Cette deuxième contribution a fait l'objet d'une publication à la *21st International Conference on Real-Time Networks and Systems (RTNS)* [LJP13d] où elle a été récompensée d'un *Oustanding Paper Award* ainsi que du *Best Student Paper Award*.

Toujours en utilisant la même approche, notre troisième contribution est un algorithme d'ordonnancement pour des systèmes temps réel à criticité mixte, c'est à dire des systèmes où les tâches de plus faible criticité peuvent tolérer des dépassements d'échéances. Cet algorithme d'ordonnancement réduit davantage la consommation énergétique en faisant un compromis entre la réduction de la consommation énergétique et le nombre de dépassements d'échéances pour les tâches de plus faible criticité. Cette contribution a donné lieu à une publication au *First workshop on Real-Time Mixed Criticality Systems (ReTiMiCS)* [LJP13a].

1.4 Plan

Ce document est divisé en cinq chapitres. Le chapitre 2 présente tout d'abord la modélisation que nous adoptons ainsi que les solutions existantes et leurs limitations. Ensuite,

les chapitres 3 et 4 détaillent respectivement la problématique et l'approche générale qui seront utilisées pour construire les algorithmes d'ordonnancement. Ceux-ci sont détaillés aux chapitres 5 et 6, respectivement pour des systèmes temps réel dur et des systèmes temps réel à criticité mixte. Chaque algorithme y est présenté et comparé aux solutions existantes. Le chapitre 7 conclut et donne quelques perspectives pour améliorer et étendre les contributions présentées.

CHAPITRE 2

Contexte - État de l'art

Sommaire

Nous avons vu au chapitre précédent notre définition des systèmes temps réel, nous présentons maintenant la modélisation que nous adoptons. Ce chapitre détaille ensuite les consommations dynamique et statique des processeurs et nous montrons que la consommation statique est maintenant plus importante que la consommation dynamique. Nous présentons par la suite les solutions matérielles que le concepteur de systèmes peut utiliser pour réduire la consommation énergétique. La dernière partie de ce chapitre discute des algorithmes d'ordonnancement existants pour réduire la consommation statique et de leurs limitations.

2.1 Représentation des systèmes temps réel

Nous définissons dans cette section la terminologie utilisée pour modéliser un système temps réel, tout d'abord les tâches puis les algorithmes d'ordonnancement. Nous définissons un système comme un ensemble de composants centrés sur une unité de calcul (i.e. un processeur) et également une mémoire ainsi que des entrées/sorties pour communiquer avec le monde extérieur. Au niveau logiciel, un système d'exploitation est nécessaire pour intégrer ces différents composants, c'est la plus basse couche logicielle s'exécutant sur le processeur. Elle est chargée de faire la liaison entre les applications du système et ses composants.

Les systèmes temps réel sont généralement séparés en plusieurs catégories suivant le niveau de sûreté temporelle demandé. Les deux catégories principales sont le temps réel dur et le temps réel mou. Un système temps réel dur est un système où aucune contrainte temporelle ne peut être violée. Chaque violation est considérée comme critique et le système ne peut fonctionner correctement en cas de violation de contrainte. Au contraire, un système temps réel mou peut tolérer que certaines contraintes temporelles ne soient pas satisfaites. Ces violations ne sont pas critiques pour le bon fonctionnement du système et peuvent seulement dégrader l'expérience utilisateur du système sans compromettre le fonctionnement global. Le temps réel mou peut par exemple être trouvé dans les systèmes multimédia qui fonctionnent correctement même si l'image ou le son sont retardés.

2.1.1 Modélisation des tâches

Notre modélisation d'un système temps réel est basée sur celle proposée par Liu et Layland [LL73]. Un système temps réel est composé d'un ensemble de tâches nommé Γ qui comprend n tâches périodiques 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 correspond à une application utilisateur du système et chaque tâche est développée par l'utilisateur du système pour répondre à un besoin. 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, le modèle le plus courant dans le monde industriel, utilisé par exemple dans le standard ARINC 653 [ARI].

Travail. 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 de l'ensemble de tâches correspond au plus petit commun multiple de toutes les périodes de l'ensemble de tâches.

Pour chaque tâche, le nombre maximal de travaux J dans une hyperpériode est égal à l'hyperpériode divisée par la période de la tâche en question. Ce nombre dépend donc de la valeur de l'hyperpériode et des valeurs des périodes des tâches :

$$J = \sum_{i=1}^{n} \frac{H}{T_i} \tag{2.1}$$

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.

WCET - AET. Le WCET (*Worst Case Execution Time*) d'une tâche est la durée maximale de l'exécution de chacun de ses travaux. Le WCET de la tâche τ_i est noté C_i et le WCET du travail j est noté $j.c$. 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 travail est connu.

Le temps d'exécution réel d'un travail (*Actual Execution Time* en anglais) est le temps utilisé par ce travail lors de l'exécution. Nous faisons l'hypothèse que l'AET est compris entre 0 et le WCET de ce travail.

Échéance. 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 $j.d$ du travail j sera donc la date de sa libération additionnée de cette échéance relative.

Nous faisons l'hypothèse que l'échéance de chaque tâche est égale à sa période. Dans la littérature, ces ensembles de tâches sont appelés ensembles de tâches à « échéances implicites ». Un autre modèle utilisé est le modèle à « échéances contraintes » où l'échéance de chaque travail est inférieure ou égale à sa période. Enfin, le modèle à « échéances arbitraires » ne fixe aucune contrainte entre les échéances et les périodes des tâches.

Criticité. Chaque tâche est caractérisée par sa criticité. Nous supposons deux niveaux de criticité : *HIGH* et *LOW*. Pour les systèmes temps réel dur, la criticité de chaque tâche est *HIGH* car aucun dépassement d'échéances n'est autorisé. Les systèmes temps réel à criticité mixte permettent d'ordonnancer des tâches de différents niveaux de criticité et les tâches de criticité *LOW* autorisent qu'un certain pourcentage d'échéances soient violées. Ce modèle est différent du modèle classique des systèmes temps réel à criticité mixte proposé par Vestal et al. [Ves07]. Nous détaillerons au chapitre 6 quels sont les différences entre le modèle que nous adoptons ici et le modèle classique.

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 $U_i = \frac{C_i}{T_i}$.

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

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.2}$$

Laxité. La laxité d'un travail est une caractéristique qui évolue lors de l'exécution et qui peut être utile à l'algorithme d'ordonnancement. A l'instant t, la laxité $j.l$ du travail j est de :

$$j.l = j.d - t - j.c(t) \tag{2.3}$$

Où $j.c(t)$ représente le temps d'exécution restant du travail j à l'instant t. Quand la laxité d'un travail atteint zéro, ce travail doit nécessairement être exécuté pour éviter un dépassement d'échéances.

Slack time. Le *slack time* libéré par un travail est la différence entre le WCET de ce travail et son AET. Comme l'AET d'un travail est compris entre 0 et le WCET de ce travail, le *slack time* libéré peut également être compris entre 0 et le WCET.

Toutes les notations relatives à l'ensemble de tâches sont résumées dans le Tableau 2.1. D'autres notations seront introduites par la suite.

Γ	Ensemble de tâches
H	Hyperpériode de l'ensemble de tâches Γ
τ_i	Tâche
C_i	WCET de la tâche i
D_i	Échéance de la tâche i
T_i	Période de la tâche i
U_i	Utilisation de la tâche i
L_i	Laxité de la tâche i
$j.c$	WCET du travail j
$j.d$	Échéance absolue du travail j
$j.l$	Laxité du travail j

Table 2.1 – Notations relatives aux tâches et travaux.

2.1.2 Caractéristiques des algorithmes d'ordonnancement multiprocesseurs

Cette section traite des algorithmes d'ordonnancement multiprocesseurs et présente les notions utilisées dans la suite de ce travail. Le lecteur est invité à consulter par exemple Davis et Burns [DB11c] pour plus d'informations sur ce sujet. Un algorithme d'ordonnancement est chargé de répartir les tâches sur les processeurs : il décide quelle tâche sera exécutée sur tel processeur et pour combien de temps.

2.1.2.1 Définitions

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 et dispose par conséquent de davantage d'informations sur les temps d'exécution des tâches du système. La complexité d'un algorithme d'ordonnancement en-ligne doit rester minimale pour laisser le temps processeur au maximum aux tâches temps réel du système.

Priorités. Les algorithmes d'ordonnancement temps réel peuvent être classés suivant leur utilisation des priorités pour choisir quelle tâche doit être ordonnancée. Une première solution est d'assigner une priorité avant l'exécution à chaque tâche et chaque travail aura ensuite la priorité de la tâche associée, c'est ce que l'on nomme priorités fixes au niveau des tâches. Les algorithmes d'ordonnancement monoprocesseur *Rate Monotonic (RM)* [LL73] et *Deadline Monotonic (DM)* [LW82] utilisent cette solution. D'autres solutions permettent de fixer les priorités au niveau des travaux (e.g. *Earliest Deadline First (EDF)* [LL73]) ou d'avoir des priorités complètement dynamiques comme *Least Laxity First (LLF)* [DM89]. Le Tableau 2.2 résume ces différentes classes d'algorithmes suivant les priorités des tâches et des travaux.

Priorités	Exemple d'algorithmes
Priorités fixes au niveau des tâches	RM [LL73], DM [LW82]
Priorités fixes au niveau des travaux	EDF [LL73]
Priorités dynamiques	LLF [DM89]

Table 2.2 – Classes d'algorithmes d'ordonnancement temps réel [Fau11].

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.

Oisiveté. Un algorithme d'ordonnancement oisif est un algorithme d'ordonnancement où un processeur peut être inactif alors qu'une ou plusieurs tâches sont susceptibles d'être exécutées. La majorité des algorithmes d'ordonnancement temps réel sont non-oisifs, ce qui permet de simplifier la conception de l'algorithme d'ordonnancement. En effet, la seule décision à prendre par l'algorithme d'ordonnancement est alors de choisir la tâche à exécuter, et non de choisir si une tâche doit être exécutée ou non.

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. Dans le cadre de ce travail, un algorithme d'ordonnancement optimal correspond à cette définition et ne signifie pas que l'algorithme d'ordonnancement est optimal au regard de la réduction de la consommation énergétique.

Borne d'ordonnançabilité. La borne d'ordonnançabilité d'un algorithme d'ordonnancement est l'utilisation globale maximale en deçà de laquelle tous les systèmes de tâches sont nécessairement ordonnançables. Pour des systèmes monoprocesseurs avec n tâches périodiques synchrones, la borne d'ordonnançabilité d'EDF est de 1, tandis que celle de RM n'est que $n(\sqrt[n]{2}-1)$ lorsque n est grand, soit environ 0.69 [LL73].

2.1.2.2 Ordonnancement de systèmes multiprocesseurs

Cette section propose un résumé non exhaustif des algorithmes d'ordonnancement multiprocesseurs existants. Ces algorithmes n'ont pas comme objectif de réduire la consommation énergétique mais il est néanmoins important d'avoir une connaissance de ces solutions car les algorithmes d'ordonnancement réduisant la consommation énergétique doivent satisfaire les contraintes d'ordonnancement. Nous supposons ici que le système possède au minimum deux processeurs. L'ordonnancement des systèmes temps réel multiprocesseurs ne peut être considéré comme la généralisation de l'ordonnancement monoprocesseur. Ce problème est plus complexe que sur systèmes monoprocesseurs car il ajoute une dimension spatiale à un problème d'allocation temporelle des ressources. L'ordonnancement multiprocesseur souffre également d'anomalies d'ordonnancement [AJ02]. Un algorithme d'ordonnancement multiprocesseur peut par exemple ordonnancer correctement un ensemble de tâches et engendrer des dépassements d'échéances lorsque la période d'une tâche de cet ensemble de tâches augmente.

Les algorithmes d'ordonnancement multiprocesseurs peuvent être divisés en plusieurs catégories selon un certain nombre de critères. Ils sont généralement triés en trois catégories selon qu'ils autorisent ou non la migration des tâches et des travaux entre processeurs.

Algorithmes partitionnés. Un algorithme d'ordonnancement partitionné ne permet pas à un travail ou à une tâche de migrer d'un processeur à un autre. Chaque tâche est assignée hors-ligne à un processeur et reste sur ce même processeur durant toute l'exécution du système. Chaque processeur peut donc être ordonnancé comme un système à un seul processeur. Allouer les tâches aux processeurs est un problème similaire au problème de *bin packing*, qui est NP-Complet [GJ90], et des heuristiques comme First-Fit ou Best-Fit peuvent être utilisées [Ber07, LW82].

Ces algorithmes sont souvent utilisés en pratique de par leur facilité d'implémentation. En effet, après l'allocation des tâches au processeur, le problème se résume à m problèmes monoprocesseurs qui sont des problèmes connus et largement étudiés. En contrepartie, ces algorithmes d'ordonnancement ont une borne d'ordonnançabilité plus faible que les algorithmes d'ordonnancement qui autorisent les migrations [Ber07].

Algorithmes globaux. Cette catégorie d'algorithmes temps réel multiprocesseurs est la plus permissive, toute migration est autorisée, que ce soit pour les travaux ou les tâches. Cette approche permet d'obtenir des algorithmes optimaux.

Global-EDF [Bak03] est un algorithme d'ordonnancement multiprocesseur global utilisant EDF pour ordonnancer les tâches. Plusieurs variantes de Global-EDF existent, nous utiliserons par la suite la variante autorisant toutes les migrations. Nous supposons que Global-EDF exécute la tâche dont l'échéance est la plus rapprochée et exécute la tâche avec l'identifiant le plus faible si deux tâches ont une même échéance.

Le premier algorithme optimal, PFair, a été proposé par Baruah et al. [BCPV93]. Le principe de cet algorithme est d'allouer du temps processeur à une tâche en fonction de son utilisation, le temps étant découpé en de courts intervalles de temps. Bien que cette première solution ait été améliorée par la suite avec *Boundary Fair (BF)* [ZMM03], cette approche n'est pas utilisable du fait du grand nombre de préemptions et de migrations engendrées. Anderson et al. [AHS05] propose un résumé des différents algorithmes d'ordonnancement reposant sur cette notion de « fairness » où chaque tâche reçoit le temps processeur qui lui est imparti en fonction de son utilisation.

Différents algorithmes d'ordonnancement temps réel multiprocesseurs globaux optimaux ne reposant pas sur la notion de fairness ont depuis été proposés, parmi lesquels *Incremental scheduling with Zero Laxity (IZL)* [MSD10], U-EDF [NBGM11, NBN+12] et *Reduction to UNiprocessor (RUN)* [RLM+11]. Cependant, tous ces algorithmes souffrent de certains défauts, comme une complexité en-ligne importante pour U-EDF ou le fait qu'ils ne puissent pas ordonnancer de systèmes de tâches sporadiques pour RUN et IZL.

D'autres algorithmes se basent sur la laxité des tâches, comme par exemple *Fixed Priority with Zero Laxity (FPZL)* [DB11a] ou *Earliest Deadline First with Zero Laxity (EDZL)* [WCL+07]. Ces algorithmes monitorent la laxité des tâches du système et ordonnancent en priorité la tâche dont la laxité est nulle pour ne pas violer son échéance.

Les approches classiques utilisent les priorités pour ordonnancer les tâches, d'autres solutions sont néanmoins possibles. Une autre méthode a par exemple été proposée par Lemerre et al. [LDAVN08]. Elle exprime le problème d'ordonnancement comme le fait d'assigner sur une hyperpériode divisée en intervalles un poids à chaque tâche sur chaque intervalle. Comme

nous allons utiliser cette solution, nous la détaillerons au chapitre 4.

Algorithmes à migrations restreintes. Entre les algorithmes partitionnés qui n'autorisent aucune migration et les algorithmes globaux où toutes les migrations sont autorisées, les algorithmes à migrations restreintes autorisent les migrations mais sous certaines conditions. La migration des tâches peut par exemple être autorisée mais pas la migration des travaux. La tâche peut donc changer de processeur lorsqu'un travail termine son exécution mais n'y est pas autorisé lorsqu'un travail est en cours d'exécution. Aucun algorithme d'ordonnancement optimal n'existe pour cette catégorie d'algorithme.

Systèmes temps réel à criticité mixte. Les algorithmes d'ordonnancement s'intéressant aux systèmes temps réel à criticité mixte ont pour objectif d'améliorer l'ordonnançabilité de ces systèmes, notamment celle des tâches à faible criticité tout en garantissant les échéances des tâche à haute criticité. Le modèle classique des systèmes temps réel à criticité mixte a été proposé par Vestal [Ves07] et définit un WCET pour chaque niveau de criticité, pour représenter par exemple les différents niveaux d'exigence des autorités de certification. Dans cette représentation, plus le niveau de criticité est élevé, plus le WCET est important. Dans le cadre de systèmes multiprocesseurs, Li et Baruah [LB12] utilisent un ordonnancement global. Mollison et al. [MEA+10] utilisent eux une stratégie partitionnée en utilisant des algorithmes d'ordonnancement différents suivant la criticité des tâches. Burns et Davis [BD13] proposent un résumé des dernières recherches sur les systèmes à criticité mixte. À notre connaissance, aucun algorithme d'ordonnancement n'a été proposé pour réduire la consommation énergétique des systèmes temps réel à criticité mixte.

2.2 Représentation de la consommation énergétique

Nous ciblons les systèmes temps réel multiprocesseurs embarqués critiques avec un nombre restreint de périphériques. Le processeur est le principal composant et il est responsable du bon fonctionnement global du système, les autres composants comme la mémoire ou les systèmes d'entrée/sortie ne pouvant communiquer qu'avec lui. Un autre point à signaler est le fait que la consommation énergétique des autres composants est fortement liée à la consommation énergétique du processeur. En effet, la mémoire du système et les périphériques doivent être actifs quand le processeur est actif. La consommation énergétique est donc principalement due aux processeurs et à leur hiérarchie de cache [CH10, SBDM99].

2.2.1 Caractéristiques des processeurs ciblés

Dans le cadre de cette thèse, nous nous intéressons aux systèmes multiprocesseurs, soit des systèmes avec plusieurs cœurs de calcul. Nous pouvons distinguer deux modélisations possibles pour les systèmes multiprocesseurs [Fau11] :

— Systèmes homogènes : tous les processeurs du système sont identiques.

— Systèmes uniformes : chaque processeur a une capacité de calcul qui lui est propre. D'autres caractéristiques peuvent être différentes mais nous restreignons dans le cadre de cette thèse à la vitesse des processeurs.

Dans le cadre de cette thèse nous traitons uniquement des systèmes homogènes où tous les processeurs ont une capacité de calcul identique. Nous parlerons de système multiprocesseurs et non de système multicœurs mais ils sont pour nous synonymes. Les hypothèses que nous faisons dans la modélisation, e.g. ignorer les échanges entre les cœurs, font que ces deux termes sont identiques.

Chaque système est également équipé de mémoire cache généralement séparé en trois niveaux. Le cache de premier niveau (L1) est plus rapide et plus petit alors que les deux niveaux suivants L2 et L3 ont une taille de plus en plus importante aux prix d'une rapidité plus faible. Sur des systèmes embarqués, la mémoire cache est en général réduite pour notamment limiter la taille de la puce. Les systèmes avec un cache L3 sont par exemple rares et des systèmes sans cache L2 existent également. Le cache L1 est lui bien présent et est propre à chaque processeur.

Les processeurs que nous ciblons sont les processeurs embarqués du commerce à base de processeurs ARM par exemple. Les processeurs à base de processeur ARM, que ce soit Cortex-A, Cortex-R ou ARM Cortex-M ont été conçus pour avoir une consommation énergétique limitée, et les Cortex-A sont maintenant disponibles en 2 ou 4 cœurs identiques [A9]. Sur ce type de processeur, tous les cœurs ont la même vitesse de calcul. Nous pouvons également citer les processeurs FreeScale basés sur une architecture PowerPC qui sont également utilisés dans la recherche sur les systèmes temps réel (e.g. [AP13]). Suivant le constructeur, les processeurs à base de cœurs ARM ont entre 1 et 4 processeurs.

La Figure 2.1 représente par exemple la topologie d'un processeur ARM Cortex-A9 avec deux processeurs. Ce processeur est équipé d'un cache L1 sur chaque processeur et d'un cache L2 commun aux deux processeurs.

2.2.2 Modèles énergétiques

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.

2.2.2.1 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

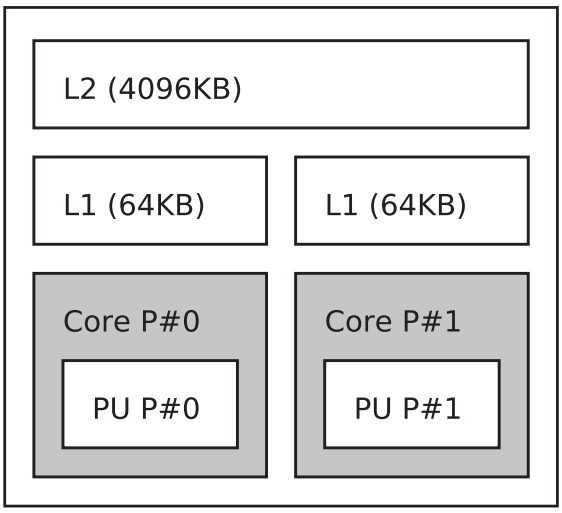

Figure 2.1 – Topologie d'un dual-core ARM Cortex-A9.

suivante [CK07] :

$$P_{dynamique} = C * f * V_{dd}^2 \tag{2.4}$$

où C correspond à la capacité de sortie du circuit et V_{dd} à la tension d'alimentation. Avec l'aide d'approximations sur le fonctionnement des circuits, il est possible de démontrer que la fréquence est proportionnelle à la tension d'alimentation. La puissance dynamique peut donc s'écrire plus simplement :

$$P_{dynamique} \propto f^3 \tag{2.5}$$

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 [WWDS94, YDS95, PS01]. Ils seront détaillés par la suite en section 2.3.

2.2.2.2 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.

Les avancées technologiques (miniaturisation, augmentation de la densité des circuits, ...) font que la consommation statique devient de plus en plus importante au détriment de la consommation dynamique. Alors que dans les années 1990 la consommation statique était souvent considérée comme négligeable, elle représente maintenant selon certaines sources [HXW+10] 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.

L'augmentation de la consommation statique a été identifiée comme un problème majeur par l'*International Technology Roadmap for Semiconductors (ITRS)* en 2010 [ITR]. La Figure 2.2 montre l'évolution des courants de fuite par rapport à la consommation dynamique depuis les années 1990 et illustre le fait que la consommation statique est maintenant plus importante que la consommation dynamique.

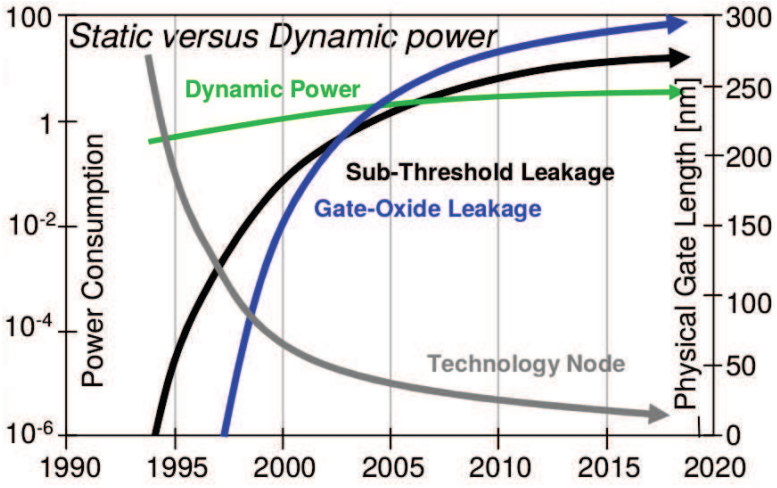

Figure 2.2 – Rapport entre consommation dynamique et courants de fuite [KAB+03].

2.2.3 Possibilités matérielles pour contrôler la consommation énergétique

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.

Comme dans la littérature existante (e.g. [AP11]), 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 :

$$Pen_s = \frac{1}{2} \times Del_s \times (Cons_{actif} - Cons_s) \tag{2.6}$$

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 période 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_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.

ns	Nombre d'états basse-consommation
$Cons_s$	Consommation énergétique de l'état basse-consommation s
Del_s	Délai de transition de l'état basse-consommation s
Pen_s	Penalité énergétique de l'état basse-consommation s
BET_s	BET de l'état basse-consommation s

Table 2.3 – Notations relatives aux états basse-consommation.

Toutes les notations introduites dans cette section sont résumées dans le tableau 2.3. Nous illustrons ces notations en Figure 2.3 qui réprésente la consommation énergétique du processeur en fonction du temps. Nous faisons l'hypothèse que le processeur possède un état basse-consommation s dont le BET est égal à 5 et le délai de transition est égal à 3.

Nous supposons qu'une tâche est exécutée à partir de $t = 0$ et termine son exécution à $t = 4$. La consommation énergétique du processeur lors de l'exécution de τ est égale $Cons_{actif}$. Le processeur doit ensuite être disponible pour exécuter une tâche à $t = 10$. 6 unités de temps sont donc disponibles pour utiliser un état basse-consommation, ce qui est supérieur au BET. L'état basse-consommation est par conséquent activé et la consommation du processeur de $t = 4$ à $t = 7$ est de $Cons_s$. À $t = 7$, le processus de réveil du processeur est enclenché pour que le processeur soit revenu à l'état actif à $t = 10$. Entre $t = 7$ et $t = 10$, la consommation énergétique du processeur évolue linéairement entre $Cons_s$ et $Cons_{actif}$.

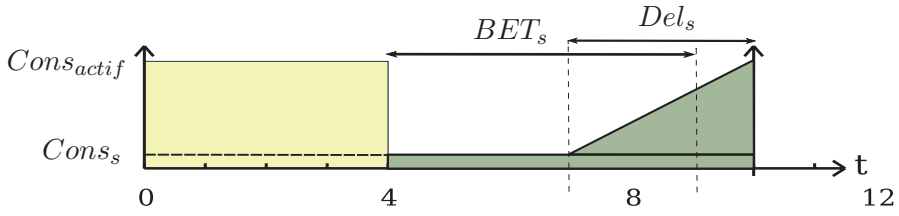

Figure 2.3 – Illustration du BET et du délai de transition.

2.2.3.1 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.

Mode	Délai de transition (μs)	BET (μs)	Consommation (W)
Maximum	-	-	12.1
Typical	0	0	4.7
Doze	5	225	3.7
Nap	100	450	2.6
Sleep	200	800	2.2
Deep Sleep	500	1400	0.6

Table 2.4 – États basse-consommation du MPC8536 [MPC, AYP13].

Dans le cadre de cette thèse, nous nous sommes inspirés des caractéristiques des états basse-consommation du STM32L [STM] de ST Microelectronics qui est basé sur un processeur ARM Cortex-M3. Nous avons fais ce choix car, contrairement à la majorité des processeurs, les consommations énergétiques et les délais de transition des états basse-consommation de ce processeur ont été rendus publiques par le constructeur. Ils sont détaillés dans le Tableau 2.5.

Ces 4 états basse-consommation sont présentés ci-dessous, en détaillant quels sont les composants éteints pour chaque état basse-consommation :

— *Run*. Mode de fonctionnement normal. Le processeur ainsi que les caches sont actifs ;

— *Sleep*. Le processeur est arrêté et tous les caches restent actifs ;

— *Low power run*. La vitesse du processeur est limitée et un nombre restreint de périphériques est disponible ;

Mode	Délai de transition (μs)	Consommation
Run		7.8 mA
Sleep	1	2.3 mA
Low power run	4	25 μA
Stop	8	3.1 μA
Standby	50	1.55 μA

Table 2.5 – États basse-consommation du STM32L [STM].

— *Stop*. Le processeur est au repos mais le contenu de la RAM et des registres est conservé. Certaines horloges sont également arrêtées. ;

— *Standby*. Tous les processeurs, les horloges et les caches sont arrêtés et le contenu de tous les caches est perdu. À noter que si plusieurs processeurs partagent le même cache L2, tous ces processeurs doivent activer ce mode simultanément.

Certains états basse-consommation désactivent les caches et leur contenu est donc perdu. Lors du retour à l'état actif, l'exécution des tâches sera par conséquent plus lente le temps que le cache retrouve son contenu. Dans le cadre de cette thèse, nous faisons l'hypothèse que le temps d'exécution des tâches ne dépend pas du contenu des caches et nous ignorons donc cette contrainte comme le font les travaux existants (e.g. [BFBA09, AP13]).

2.3 État de l'art

Cette section présente les principaux travaux existants pour réduire la consommation énergétique des systèmes temps réel. Nous nous intéressons tout d'abord aux travaux se concentrant sur la consommation dynamique puis ensuite aux travaux essayant de réduire la consommation statique pour des systèmes monoprocesseurs puis multiprocesseurs.

2.3.1 Réduction de la consommation dynamique

Nous proposons ici un aperçu des solutions développées pour réduire en priorité la consommation dynamique, pour des systèmes monoprocesseurs puis multiprocesseurs. Ces travaux sont généralement antérieurs aux travaux se focalisant sur la consommation statique car la consommation dynamique était alors plus importante. Sauf si le modèle de tâches est explicitement mentionné, nous supposons dans cette section que nous avons des modèles de tâches sporadiques où la période d'une tâche est la période de temps minimale entre deux libérations de travaux.

2.3.1.1 Systèmes monoprocesseurs

Weiser et al. [WWDS94] ainsi que Yao et al. [YDS95] ont été les premiers à avoir proposé des approches permettant de réduire l'énergie consommée par le processeur. Yao et al. ont proposé deux algorithmes permettant de réduire l'énergie consommée par le processeur :

le premier est hors-ligne et est optimal tandis que le second est en-ligne. Hors-ligne, leur algorithme identifie les intervalles critiques où la charge processeur est la plus importante et assigne une fréquence minimale de fonctionnement permettant le respect des échéances. Ces premiers résultats sur la réduction de la consommation énergétique des systèmes temps réel ont par la suite inspiré de nombreux travaux (e.g. [HPS98, SCS00, PS01, GN07]) que nous ne développerons pas ici pour nous concentrer sur les solutions permettant de réduire la consommation statique. Pour plus d'informations, nous renvoyons le lecteur à Navet et Gaujal [NG06] qui en proposent un résumé.

Une autre idée développée par Mosse et al. [MACM00] consiste à insérer des *Power Management Points* dans le code de l'application. Une échéance étant associée à chaque point, il est possible de connaître lors de l'exécution du programme si le temps d'exécution va se rapprocher de l'échéance ou si du temps processeur va être disponible pour abaisser la vitesse du processeur. Cette information est ensuite traitée dynamiquement et la vitesse du processeur est modifiée pour être la plus faible possible tout en assurant que l'application respecte ses échéances. L'ajout de ces points dans le code de l'application permet d'obtenir des informations sur l'avancée de l'exécution de la tâche et ainsi de gérer plus finement la vitesse du processeur. La notion de *Power Management Points* a été utilisée et développée par AouGhazaleh et al. [AMCM03] et Melhem et al. [MAAM02].

Aydin et al. [AMAMM01], dans le cas de tâches périodiques, proposent une solution en trois parties pour minimiser la consommation énergétique d'un système temps réel monoprocesseur où la fréquence du processeur peut varier de façon continue (en donnant l'exemple du processeur Transmeta Crusoe dont la fréquence varie par plage de 33 MHz). Après avoir calculé hors-ligne la vitesse optimale du processeur en prenant le pire cas d'exécution des tâches, deux algorithmes en-ligne sont proposés. Le premier permet de réduire la vitesse en prenant en compte le temps d'exécution réel des tâches et le second utilise le temps moyen d'exécution pour anticiper la fin prochaine de l'exécution d'une tâche toujours dans le but de réduire la fréquence du processeur. Une étude similaire a été menée par Krishna et Lee [KL00].

2.3.1.2 Réduction de la consommation énergétique globale du système

Aydin et al. [ADZ06] ont pris en compte la totalité du système, le processeur mais également les périphériques dont la consommation énergétique dépend de la charge de travail du processeur. D'autres solutions ont été également proposées pour prendre en compte la consommation énergétique des autres composants du système et plus particulièrement les périphériques d'entrée/sortie [DA08a, DA08b, KWDY10, SC05, CG06, QZ08].

2.3.1.3 Systèmes multiprocesseurs

Pour Anderson et Baruah [AB04], il est plus efficace d'utiliser DVFS sur plusieurs processeurs du fait de la relation convexe entre la puissance et la consommation dynamique. Aydin et Qi [AY03] ont été les premiers à proposer une solution pour systèmes multiprocesseurs. Ils proposent une solution partitionnée pour réduire au mieux la vitesse du processeur. Ils montrent également que le problème est NP-difficile au sens fort. Bhatti et al. [BBA10c] ainsi que Nelis et al. [NGD+08, NG09, Né10] utilisent eux des approches globales pour réduire la

consommation dynamique. Pour des systèmes hétérogènes où les processeurs du système ne sont pas identiques, Hung et al. [HCK06] ont pour leur part associé un processeur dont la fréquence est variable avec un composant à fréquence fixe dont la consommation énergétique peut dépendre ou non de la charge de travail.

Chen et Kuo [CK07] ont proposé un résumé des algorithmes d'ordonnancement temps réel monoprocesseurs et multiprocesseurs dont l'objectif est de réduire la consommation dynamique.

2.3.2 Réduction des consommations dynamique puis statique

Dans cette section nous nous intéressons aux travaux dont le but est de réduire la consommation statique, mais seulement après avoir réduit la consommation dynamique, soit seulement lorsque réduire la fréquence du processeur n'est plus possible. Cette situation arrive lorsque la fréquence désirée n'est pas disponible sur le processeur utilisé, le processeur est alors dans l'obligation de fonctionner à une fréquence plus élevée et du temps processeur est ainsi libéré. Ces solutions ont pour objectif d'utiliser ce temps processeur pour activer un état basse-consommation. Le principal défaut de ces solutions est que les périodes d'inactivité sont alors relativement courtes. La fréquence en deçà de laquelle il n'est plus possible d'aller est souvent nommée dans la littérature *fréquence critique*.

Pour s'adapter aux différentes charges de travail des différentes applications, Dhiman et Rosing [DR06] ont développé une méthode d'apprentissage dont le but est de reconnaitre la charge de travail en cours afin d'utiliser l'état basse-consommation le plus adapté.

Cette solution a été étendue aux systèmes multiprocesseurs en utilisant à la fois des algorithmes réduisant la fréquence de fonctionnement et des états basse-consommation par Bhatti et al. [BBA11]. Leur solution, *Hybrid Power Management scheme*, cible les systèmes multiprocesseurs composés de tâches à échéances implicites. Elle est basée sur le constat que les politiques d'ordonnancement, qu'elles utilisent DVFS ou les états basse-consommation, ne s'adaptent pas à toutes les conditions de fonctionnement et qu'il serait plus efficace de choisir la meilleure solution en-ligne. Pour savoir quel algorithme utiliser, leur solution calcule lors de l'exécution la consommation énergétique que chaque algorithme d'ordonnancement aurait engendrée et choisit celle dont la consommation énergétique est la plus faible sur la dernière fenêtre temporelle. Cette solution est trop complexe pour être mise en œuvre car elle implique de calculer lors de l'exécution la consommation énergétique de tous les algorithmes d'ordonnancement.

Irani et al. [IGS02, ISG03a, ISG03b, IP05] cherchent également à réduire et la puissance dynamique et la puissance statique en réduisant la fréquence du processeur ou en utilisant un état basse-consommation. La solution de Yao et al. [YDS95] est utilisée pour trouver une vitesse à tous les travaux. Si la vitesse est supérieure à la vitesse critique, le travail est ordonnancé normalement. Si la vitesse est inférieure à la vitesse critique, l'objectif est d'augmenter la vitesse d'exécution pour arriver à la vitesse critique, ce qui dégage un temps processeur où il est possible d'activer les états basse-consommation. Le but est de bien organiser ces travaux pour économiser de l'énergie ou réduire le nombre de périodes d'inactivité où le processeur est inactif.

Awan et Petters [AP13] ont proposé une approche hors-ligne partitionnée pour un système de tâches sporadiques à échéances implicites. Ils utilisent des systèmes multiprocesseurs hétérogènes où chaque processeur a une fréquence différente. Comme tous les processeurs sont différents, ils font l'hypothèse que chaque tâche possède une consommation énergétique différente sur chaque processeur. Leur approche est en deux parties, ils placent tout d'abord les tâches sur les processeurs en fonction de l'efficacité énergétique de chaque tâche sur chaque processeur. Ensuite, ils prennent en compte les états basse-consommation pour réassigner les tâches sur d'autres processeurs si des états basse-consommation plus profonds peuvent être utilisés. Comme tous les algorithmes d'ordonnancement présentés dans cette section, cet algorithme d'ordonnancement ne se focalise donc sur la consommation statique que dans un second temps.

2.3.3 Réduction de la consommation statique

Nous nous intéressons dans cette section aux solutions qui ont comme objectif premier la réduction de la consommation statique des systèmes temps réel. Tout d'abord pour les systèmes monoprocesseurs puis pour les systèmes multiprocesseurs, avec des approches partitionnées puis globales.

2.3.3.1 Systèmes monoprocesseurs

Pour réduire la consommation énergétique des systèmes monoprocesseurs, Benini et al. [BBBM00] ont été les premiers à proposer des algorithmes d'ordonnancement pour activer les états basse-consommation des processeurs. Cependant, les différentes solutions qu'ils proposent ne sont pas compatibles avec le temps réel dur. En effet, ils activent les états basse-consommation sans connaître au préalable la taille de la période d'inactivité à venir.

Sur des systèmes monoprocesseurs et pour des systèmes de tâches sporadiques à échéances implicites, Lee et al. [LRK03] ont les premiers proposé un algorithme pour augmenter la taille des périodes d'inactivité. Quand le processeur est inactif, leur idée est de retarder l'exécution des tâches pour garder le processeur dans un état basse-consommation le plus longtemps possible. Quand une tâche est libérée alors que le processeur est dans un état basse-consommation, l'intervalle maximal durant lequel il est possible de laisser le processeur au repos est calculé afin d'éviter toute violation d'échéances. La complexité de cette solution pour calculer la taille maximale de cet intervalle est en $O(n^2)$ où n est le nombre de tâches.

D'autres solutions ont ensuite été proposées pour améliorer ce calcul, par Jejurikar et al. [JPG04, JG05], Chen et Kuo [CK06] et Niu et Quand [NQ04]. Zhu et al. [ZM07] ont également proposé une amélioration lorsque les tâches n'utilisent pas leur WCET.

Cependant, la plupart de ces solutions utilisent les états basse-consommation seulement en complément de DVFS. Leur objectif principal est de réduire la consommation dynamique et ils ne s'intéressent à la consommation statique que lorsque la fréquence ne peut plus être réduite. En prenant en compte seulement leur solution pour étendre la taille des périodes d'inactivité, le défaut principal de ces solutions monoprocesseurs est leur complexité en-ligne pour calculer la taille maximale de la période d'inactivité. Cela nécessite en effet de retarder l'exécution des tâches. Par exemple, la complexité de la solution proposée par Niu et al. [NQ04] est en $O(J^3)$

où J est le nombre de travaux dans l'hyperpériode. A noter toutefois que cette solution est optimale dans le sens où augmenter la taille de la période d'inactivité aboutit nécessairement à un dépassement d'échéances.

Awan et al.[AYP13] ont récemment proposé une solution permettant de calculer la taille maximale de l'intervalle avec une complexité réduite en $O(nJ)$ par rapport à Niu et al.[NQ04] en utilisant le principe de *Demand Bound Function* (DBF) [BRH90].

Une autre approche a été suivie par Awan et Petters [AP11]. Ils utilisent un modèle composé de tâches temps réel dur et temps réel mou. Ils calculent hors-ligne la taille maximale de la période d'inactivité pour réduire la complexité en-ligne. Is utilisent le *slack time* libéré lorsque les tâches n'utilisent pas leur WCET pour augmenter le temps passé dans un état basse-consommation.

En conclusion, il existe des solution efficaces pour réduire la consommation statique des systèmes temps réel monoprocesseurs en optimisant la taille des périodes d'inactivité pour activer les états basse-consommation les plus efficaces. Nous détaillons maintenant les solutions existantes pour les systèmes multiprocesseurs.

2.3.3.2 Systèmes multiprocesseurs avec approche partitionnée

Les systèmes multiprocesseurs ont été moins étudiés que les systèmes monoprocesseurs, et les approches partitionnées ont été privilégiées.

Pour des tâches périodiques à échéances implicites, Chen et al. [CHK06] proposent deux algorithmes d'ordonnancement en utilisant une approche partitionnée pour réduire les consommations dynamique puis statique. Leurs algorithmes affectent tout d'abord chaque tâche à un processeur et utilisent ensuite des algorithmes monoprocesseurs pour réduire la consommation énergétique sur chaque processeur. L'algorithme *LA + LTF* assigne les tâches aux processeurs en utilisant une stratégie *Largest-Task-First*. Les tâches sont triées par utilisations décroissantes et assignées une à une au processeur dont l'utilisation est la plus faible. Les tâches sont ensuite exécutées en utilisant EDF sur chaque processeur. L'algorithme *LA + LTF + FF* utilise tout d'abord l'algorithme *LA + LTF* puis réassigne les tâches aux processeurs si la fréquence minimale qu'un processeur peut utiliser est inférieure à la fréquence critique de ce processeur. Leur dernière optimisation nommée *PROC* est en-ligne et utilise une solution basée sur Lee et al. [LRK03] pour optimiser la taille des périodes d'inactivité sur chaque processeur. La consommation statique n'est donc réduite qu'une fois la consommation dynamique ne peut plus être abaissée. Leurs simulations montrent que l'utilisation de ces trois méthodes simultanément engendre la consommation énergétique la plus faible. Mais ces simulations ne comparent que les solutions présentées dans cet article et ne prennent pas en compte une solution qui n'aurait utilisée que les états basse-consommation pour réduire en priorité la consommation statique. À noter également que les simulations n'utilisent pas des processeurs existants mais des modèles approchés pour les consommations dynamique et statique.

Seo et al. [SJPL08] ainsi que Haung et al. [HXW+10] utilisent également une approche partitionnée mais autorisent néanmoins la migration des tâches. Leur raisonnement pour autoriser les migrations est de permettre de regrouper des périodes d'inactivité qui autrement

se seraient créées sur des processeurs différents. Le problème de cette approche est la complexité liée au fait de faire un nouveau partitionnement de l'ensemble de tâches en-ligne.

Seo et al. [SJPL08] utilisent un système de tâches périodiques à échéances implicites et ciblent à la fois la consommation dynamique et la consommation statique en privilégiant la réduction de la consommation dynamique. Ils partitionnent tout d'abord les tâches en utilisant des heuristiques comme *Best-Fit Decreasing* ou *Next-Fit Decreasing*. Chaque processeur est ordonnancé en utilisant EDF et la consommation dynamique est réduite en utilisant l'algorithme *cycle-conserving* proposé par Pillai et Shin [PS01] sur chaque processeur. Lorsqu'une tâche termine son exécution, leur algorithme *Dynamic Repartitioning* réassigne en-ligne les tâches aux processeurs pour diminuer la consommation dynamique en réduisant la fréquence des processeurs. Ensuite, pour réduire la consommation dynamique, leur algorithme *Dynamic Core Scaling* essaie lorsqu'une tâche termine son exécution de réassigner les tâches pour permettre à un processeur d'activer un état basse-consommation. Ces deux algorithmes sont utilisés simultanément en donnant la priorité au premier permettant de réduire la consommation dynamique. Leurs simulations montrent que ces algorithmes permettent de réduire la consommation énergétique jusqu'à 30% dans le meilleur des cas. Cette situation intervient lorsque l'utilisation globale est la plus faible et lorsque le nombre de processeurs est le plus élevé ce qui permet à *Dynamic Core Scaling* de désactiver le plus grand nombre de processeurs.

2.3.3.3 Systèmes multiprocesseurs avec approche globale

Il n'existe à notre connaissance aucun algorithme d'ordonnancement temps réel multiprocesseur optimal dont l'objectif soit de réduire la consommation statique. Le seul algorithme d'ordonnancement non-optimal est AsDPM [BFBA09] proposé par Bhatti et al. pour des systèmes de tâches périodiques à échéances implicites. Leur objectif est de n'utiliser qu'un seul processeur et de n'activer les processeurs restants qu'uniquement lorsqu'une échéance est sur le point d'être violée. Leur solution vient en complément d'un algorithme d'ordonnancement existant comme EDF ou LLF et n'est donc pas optimal.

Pour savoir si une tâche peut être retardée, la laxité de chaque tâche est calculée à chaque décision d'ordonnancement. Cette laxité dépend de toutes les tâches de plus haute priorité. Et si la laxité d'une tâche est inférieure à zéro, cette tâche doit être ordonnancée pour qu'aucune échéance ne soit violée. La tâche ayant la priorité la plus importante est alors ordonnancée et il faut recalculer la laxité de toutes les tâches en prenant en compte que cette tâche est ordonnancée. Chaque décision d'ordonnancement nécessite un calcul en $O(n^3)$ pour retarder les futures exécutions sans violer d'échéances ce qui rend l'algorithme non utilisable en pratique (n tâches).

Les simulations effectuées montrent que la consommation énergétique d'AsDPM est jusqu'à 18% plus faible avec AsDPM, avec respectivement un gain maximal de 16% en utilisant EDF et de 18% en utilisant LLF. Le nombre de périodes d'inactivité est lui réduit de 75% et 66% pour respectivement EDF et LLF.

AsDPM a ensuite été amélioré par Dandrimont et Jan [DJ12] pour réduire le nombre de migrations et réduire la fréquence lorsqu'une tâche ne consomme pas son WCET et libère du slack time.

2.4 Conclusion

Ce chapitre a présenté les systèmes que nous ciblons dans le cadre de cette thèse ainsi que le vocabulaire et la modélisation que nous adoptons. La modélisation est basée sur celle utilisée dans la littérature existante et tient également compte des capacités des processeurs qui ont été mises à jour par rapport aux travaux existants.

Nous avons également passé en revue les solutions existantes pour réduire la consommation énergétique des systèmes embarqués multiprocesseurs temps réel. La littérature ne s'est que peu intéressée à réduire la consommation énergétique des systèmes temps réel multiprocesseur, les solutions existantes privilégient trop souvent la consommation dynamique au détriment de la consommation statique qui est devenue prépondérante ces dernières années.

Les solutions ayant pour objectif de réduire la consommation statique se concentrent principalement sur les systèmes monoprocesseurs, les systèmes multiprocesseurs n'ont que peu été étudiés. Parmi les algorithmes d'ordonnancement multiprocesseurs existants, seul AsDPM utilise une approche globale mais n'est pas un algorithme d'ordonnancement optimal. A notre connaissance, il n'existe aucun algorithme d'ordonnancement temps réel multiprocesseur optimal donc l'objectif soit de créer de larges périodes d'inactivité pour exploiter les états basse-consommation des processeurs.

Le prochain chapitre expose en détail les problèmes à résoudre pour réduire la consommation énergétique en utilisant les états basse-consommation des processeurs et les chapitres suivants présentent nos contributions, à savoir des algorithmes d'ordonnancement multiprocesseurs optimaux pour réduire la consommation énergétique.

CHAPITRE 3

Problématiques relatives à la réduction de la consommation statique

Sommaire

Ce chapitre présente les problèmes liés à l'activation des états basse-consommation pour réduire la consommation statique tout en respectant les contraintes temporelles. Nous présentons les caractéristiques nécessaires aux algorithmes d'ordonnancement pour résoudre ce problème et les heuristiques utilisées par les algorithmes d'ordonnancement existants. Nous détaillons les différentes possibilités suivant les cadres applicatifs visés, c'est-à-dire les systèmes temps réel dur et les systèmes temps réel à criticité mixte. Ces différents problèmes seront ensuite traités dans les chapitres suivants, le chapitre 4 présentant l'approche générale tandis que les chapitres 5 et 6 traitent respectivement des systèmes temps réel dur et des systèmes temps réel à criticité mixte.

3.1 Exploitation des états basse-consommation

Notre objectif est de réduire la consommation énergétique des systèmes temps réel embarqués. Nous avons vu au chapitre 2 que cette consommation énergétique pouvait être divisée en consommation statique et consommation dynamique. La consommation statique tend à devenir plus importante que la consommation dynamique. Nous nous focalisons donc essentiellement sur la consommation statique en activant les états basse-consommation des processeurs. Les caractéristiques de ces états basse-consommation ont été détaillées au chapitre 2. Dans un état basse-consommation, la consommation énergétique du processeur

est réduite par rapport à l'état actif, la consommation dynamique étant même nulle car le processeur est inactif. Nous supposons que la transition pour passer de l'état actif à un état basse-consommation est immédiate au contraire de la transition d'un état basse-consommation à l'état actif qui nécessite un délai de transition et une pénalité énergétique différents pour chaque état basse-consommation [HXW+10]. Les processeurs possèdent chacun plusieurs états basse-consommation, le délai de transition et la pénalité énergétique d'un état basse-consommation étant plus importants si sa consommation énergétique est plus faible.

3.1.1 Prérequis sur le modèle de tâches et les capacités matérielles

Du point de vue de l'algorithme d'ordonnancement, utiliser au mieux les états basse-consommation signifie minimiser la consommation énergétique lorsque le processeur est dans un état basse-consommation. Pour cela, les états basse-consommation les plus efficaces doivent être utilisés, c'est-à-dire les états basse-consommation qui requièrent un délai de transition plus important.

Quand le système termine l'exécution d'une tâche sur un processeur et qu'aucune autre tâche n'est prête à être exécutée sur ce même processeur, le système peut activer un état basse-consommation sur le processeur en question. Le système connaît les caractéristiques de chaque état basse-consommation, i.e. sa consommation énergétique et son délai de transition pour revenir à l'état actif. Il peut donc en fonction activer l'état basse-consommation le plus adapté à la période d'inactivité à venir.

Pour choisir l'état basse-consommation le plus efficace, le processeur doit connaître la taille de la période d'inactivité, ou au minimum avoir des garanties sur sa taille minimale. En effet, il est impossible d'activer un état basse-consommation dont le délai de transition est inférieur à la taille minimale prévue de la période d'inactivité (i.e. le BET défini au chapitre 2), sous peine de violer une échéance car le processeur ne pourra pas revenir à l'état actif à temps. Si la taille de la période d'inactivité est sous-estimée, il doit alors activer un état basse-consommation moins efficace.

Lors de l'exécution, suivant les temps d'exécution des tâches, le système peut décider d'augmenter la taille de la période d'inactivité. Nous supposons alors qu'il lui est possible de mettre à jour la date de réveil depuis un autre processeur. Nous faisons donc l'hypothèse que le système peut agir sur tous les processeurs quel que soit le processeur sur lequel il est actif. Ainsi, aucun composant supplémentaire n'est nécessaire pour éteindre et réveiller les processeurs. Cette hypothèse est correcte pour les processeurs actuels qui permettent à un processeur d'agir sur le comportement des autres processeurs comme vu au chapitre 2.

Certains travaux (e.g. [LRK03, JPG04]) font l'hypothèse qu'un composant extérieur est nécessaire pour calculer la nouvelle taille des périodes d'inactivité et mettre à jour la date de réveil du processeur. Mais ces travaux utilisent des ensembles de tâches sporadiques et ne connaissent donc que la date minimale des prochains temps de réveil des travaux. Par conséquent, pour ne pas avoir à réveiller le processeur trop tôt lors de l'activation d'une tâche, l'ajout d'un autre composant pour calculer la date de réveil du processeur est nécessaire. Comme nous travaillons avec des tâches périodiques, nous n'avons pas besoin de cette hypothèse. A noter que ces travaux utilisent des systèmes monoprocesseurs mais la

problématique est identique pour les systèmes multiprocesseurs car tous les processeurs du système peuvent lors de l'exécution être simultanément dans un état basse-consommation.

3.1.2 Propriétés nécessaires aux algorithmes d'ordonnancement

Activer les états basse-consommation les plus efficaces nécessite donc que l'algorithme d'ordonnancement optimise la taille des périodes d'inactivité. Cette section introduit deux propriétés nécessaires à ces algorithmes d'ordonnancement pour y parvenir : l'autorisation des migrations des tâches et l'oisiveté.

3.1.2.1 Autorisation des migrations

Les algorithmes d'ordonnancement multiprocesseurs partitionnés n'autorisent pas les migrations des tâches entre processeurs lors de l'exécution, et ne peuvent par conséquent générer de périodes d'inactivité aussi larges que les algorithmes globaux où les migrations sont autorisées. En effet, comme chaque processeur est ordonnancé comme un système monoprocesseur indépendant, des périodes d'inactivité sont créées sur chaque processeur. Il est donc impossible de regrouper ces périodes d'inactivité sur un seul processeur pour diminuer leur nombre et augmenter leur taille. Une solution d'ordonnancement visant à minimiser la consommation statique doit par conséquent autoriser les migrations, que ce soit de manière restreinte ou totale.

Pour illustrer ce point, l'ensemble de tâches 1 du Tableau 3.1 est ordonnancé sur la Figure 3.1 à l'aide d'un algorithme partitionné utilisant EDF sur chaque processeur tandis que la Figure 3.2 utilise l'algorithme Global-EDF. En utilisant une approche partitionnée où τ_1 est assignée au processeur π_1 et τ_2 à π_2, cinq périodes d'inactivité sont générées dans l'hyperpériode tandis qu'une approche globale comme Global-EDF permet de n'en créer que quatre.

	τ_1	τ_2
WCET	2	3.5
Période	4	6

Table 3.1 – Ensemble de tâches 1.

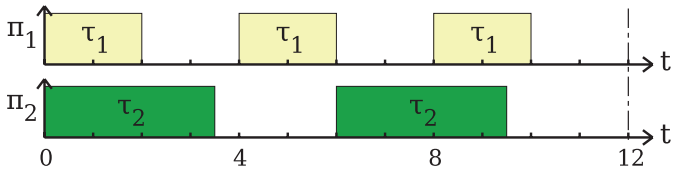

Figure 3.1 – Ordonnancement de l'ensemble de tâches 1 avec migrations interdites.

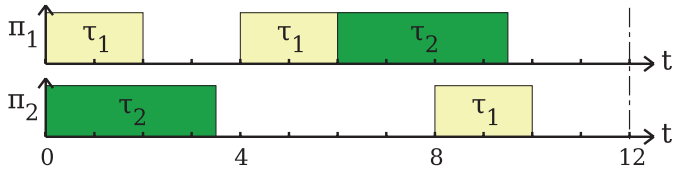

Figure 3.2 – Ordonnancement de l'ensemble de tâches 1 avec migrations autorisées.

3.1.2.2 Oisiveté

Un algorithme d'ordonnancement oisif permet à un processeur d'être inactif même si une ou plusieurs tâches sont prêtes à être exécutées. Pour que les tâches soient ordonnancées de telle sorte qu'il soit possible d'activer les états basse-consommation les plus économes en énergie, il est nécessaire que l'algorithme d'ordonnancement soit oisif. En effet, un algorithme d'ordonnancement non oisif multiprocesseur tel que Global-EDF exécute les tâches dès qu'elles sont disponibles, alors qu'il pourrait être plus judicieux de retarder l'exécution d'une tâche pour regrouper cette exécution avec l'exécution d'autres tâches et ainsi générer moins de périodes d'inactivité.

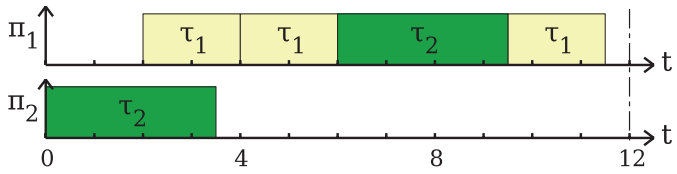

Figure 3.3 – Ordonnancement de l'ensemble de tâches 1 avec oisiveté.

L'ensemble de tâches 1 est ordonnancé sur la Figure 3.3 de manière oisive tandis que sur les deux précédentes Figures 3.1 et 3.2 utilisaient des solutions non oisives. De manière intuitive, nous avons retardé l'exécution du premier travail de τ_1 de deux unités de temps pour supprimer la période d'inactivité après le premier travail de τ_1. Nous avons également retardé le dernier travail de τ_1 pour supprimer une autre période d'inactivité. Cet exemple montre qu'autoriser l'oisiveté permet de supprimer des périodes d'inactivité donc de créer de plus larges périodes d'inactivité.

Nous avons vu dans cette section comment sont activés les états basse-consommation ainsi que deux propriétés nécessaires à l'algorithme d'ordonnancement pour optimiser la taille des périodes d'inactivité : l'autorisation des migrations et l'oisiveté. La section suivante détaille les problèmes liés à l'optimisation de la taille de ces périodes d'inactivité du point de vue de l'algorithme d'ordonnancement.

3.2 Simplification pour la génération de l'ordonnancement

L'algorithme d'ordonnancement doit optimiser la taille des périodes d'inactivité pour activer les états basse-consommation les plus efficaces et réduire la consommation statique.

Nous posons tout d'abord le problème de façon optimale et nous proposons par la suite plusieurs simplifications possibles. Nous faisons l'hypothèse dans le cadre de cette section que toutes les tâches du système consomment leur WCET. L'optimalité d'un algorithme d'ordonnancement du point de vue de la consommation énergétique est définie ainsi :

Définition 1. *Un algorithme d'ordonnancement temps réel est optimal du point de vue de la consommation énergétique lorsqu'il n'existe aucun autre algorithme d'ordonnancement ayant la même ordonnançabilité générant une consommation énergétique plus faible.*

Un tel algorithme d'ordonnancement optimal n'est pas concevable en pratique. En effet, comme vu au chapitre 2, l'ordonnancement des systèmes multiprocesseurs est un problème présentant une complexité importante et ne peut être considéré comme une généralisation de l'ordonnancement monoprocesseur. Trouver une partition d'un ensemble de tâches sur plusieurs processeurs est par exemple un problème similaire à celui du *bin packing* qui est un NP-Complet [Ber07, LW82].

Cette importante complexité est nécessaire pour générer un ordonnancement valide respectant les échéances des tâches. Or nous ajoutons ici un objectif supplémentaire qui est de réduire la consommation énergétique, donc tout algorithme d'ordonnancement visant à réduire la consommation énergétique sera plus complexe que son homologue sans cet objectif. Nous avons également vu que l'algorithme doit être oisif ce qui augmente davantage sa complexité. Cette affirmation est confirmée par les algorithmes existants comme AsDPM [BFBA09] que nous avons détaillé au chapitre 2 et dont la complexité est en $O(n^3)$ à chaque instant d'ordonnancement, avec n correspondant au nombre de tâches.

De nombreux algorithmes d'ordonnancement ont donc été proposés en utilisant des heuristiques, la première simplification étant de réduire le problème à une période de temps donnée, à savoir l'hyperpériode (e.g. [NQ04]). Une autre heuristique utilise la taille des périodes d'inactivité et leur nombre au lieu de la consommation énergétique (e.g. [LRK03]) et la troisième ne se focalise que sur la période d'inactivité courante.

3.2.1 Simplification sur une hyperpériode

Une première simplification est de ne s'intéresser qu'à une durée d'exécution limitée : l'hyperpériode. Cette simplification est habituelle dans la littérature (e.g. [NQ04]) mais a néanmoins des conséquences d'un point de vue énergétique. En effet, cette simplification n'est pas correcte car une période d'inactivité peut être comprise sur deux hyperpériodes lors de l'exécution. Donc obtenir une consommation énergétique minimale sur une hyperpériode ne permet pas d'affirmer que la consommation énergétique sera minimale sur toute la durée d'exécution.

Il est néanmoins possible de calculer un ordonnancement sur plusieurs hyperpériodes pour inclure les transitions entre les hyperpériodes jusqu'à obtenir un comportement périodique. Mais pour rester dans des limites raisonnables au niveau des temps de calcul, nous nous limiterons dans le cadre de cette thèse à une hyperpériode. Nous avons constaté empiriquement que la pénalité en complexité et en temps de calcul est trop importante pour les gains en énergie possibles. Nous supposons donc que générer un ordonnancement sur une hyperpériode est suffisant.

La complexité du problème même sur une hyperpériode peut être illustrée par une étude sur le nombre d'ordonnancements possibles. En supposant que les décisions d'ordonnancement ne se prennent que lorsqu'un travail termine son exécution ou devient disponible, la complexité dépend du nombre de travaux sur l'intervalle d'étude qui est ici l'hyperpériode. Nous avons défini J au chapitre 2 comme le nombre maximal de travaux sur une hyperpériode. Si nous faisons l'hypothèse que les travaux ne peuvent être exécutés que lorsqu'une tâche est libérée ou termine son exécution, le nombre d'ordonnancements possibles est alors de $(2 \times J)^n$, $2 \times J$ étant le nombre maximal d'instants d'ordonnancement, soit le nombre de fois où un travail apparaît plus le nombre de fois où un travail termine son exécution. Et à chaque instant d'ordonnancement, le système choisi parmi n tâches celle à exécuter. Une fois tous ces ordonnancements obtenus, il est nécessaire de calculer la consommation énergétique de toutes les périodes d'inactivité pour choisir l'ordonnancement permettant de minimiser la consommation énergétique. Si le nombre de périodes d'inactivité est de P, P^{ns} combinaisons sont possibles par ordonnancement, avec ns le nombre d'états basse-consommation introduit au chapitre 2.

3.2.2 Calcul de la taille des périodes d'inactivité

Au lieu de s'intéresser à la consommation énergétique en elle même, les algorithmes d'ordonnancement multiprocesseurs cherchant à réduire la consommation statique s'intéressent en grande majorité à la taille des périodes d'inactivité [BFBA09, CHK06, SJPL08]. En effet, plus la taille des périodes d'inactivité est importante, plus il est possible d'activer des états basse-consommation les plus économes en énergie. L'algorithme d'ordonnancement ne s'intéresse plus à la consommation énergétique mais à la taille des périodes d'inactivité ou à leur nombre. Il n'a alors plus à connaître les caractéristiques des états basse-consommation et peut générer des périodes d'inactivité dont la taille n'est pas adaptée aux états basse-consommation disponibles sur les processeurs.

Pour illustrer la signification de cette simplification, prenons deux ordonnancements possibles de l'ensemble de tâches 2 du Tableau 3.2 représentés sur les Figures 3.4 et 3.5. Sur ces deux figures, deux périodes d'inactivité sont créées sur l'hyperpériode, mais les tailles de ces périodes d'inactivité sont différentes. Nous supposons dans cet exemple que nous avons deux processeurs ayant chacun deux états basse-consommation dont les BET ont pour valeur 1 et 5. Dans la première figure, le deuxième état basse-consommation ne pourra être activé dans aucune des périodes d'inactivité. Au contraire, dans la seconde figure, la deuxième période d'inactivité permet d'activer le deuxième état basse-consommation. La consommation énergétique de l'ordonnancement de la Figure 3.5 sera donc inférieure alors que le nombre de périodes d'inactivité est identique pour les deux figures.

	τ_1	τ_2	τ_3
WCET	2	1	2
Période	3	4	4

Table 3.2 – Ensemble de tâches 2.

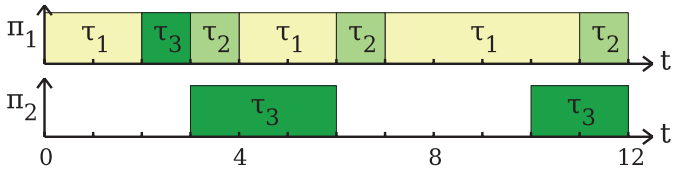

Figure 3.4 – Exemple d'ordonnancement minimisant le nombre de périodes d'inactivité.

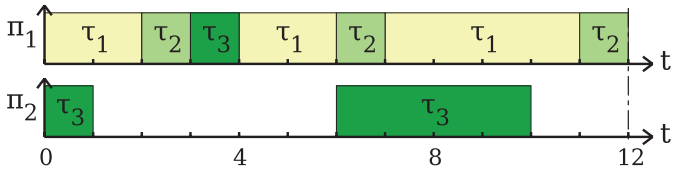

Figure 3.5 – Exemple d'ordonnancement minimisant la consommation énergétique.

Cette deuxième simplification s'intéresse au nombre de périodes d'inactivité sur une hyperpériode. Mais cela nécessite que l'algorithme d'ordonnancement anticipe les futures exécutions des tâches pour prévoir le nombre de périodes d'inactivité sur l'hyperpériode et prendre ces décisions d'ordonnancement. Or anticiper le futur est une opération complexe que les algorithmes d'ordonnancement veulent éviter, surtout pour les algorithmes d'ordonnancement en-ligne. Par conséquent, la dernière simplification qui est utilisée consiste à ne s'intéresser qu'à la prochaine période d'inactivité.

3.2.3 Optimisation de la taille de la prochaine période d'inactivité

Une simplification utilisée dans la littérature est de maximiser la taille de la période d'inactivité courante, sans s'occuper des autres périodes d'inactivité qui vont apparaître durant l'exécution du système. Une solution a été proposée pour des systèmes monoprocesseurs par Niu et al. [NQ04]. Ils ont montré que leur solution est optimale dans le sens où la taille de la période d'inactivité courante est maximale tout en garantissant qu'aucun dépassement d'échéances n'ait lieu. La complexité de la solution proposée par Niu et al. [NQ04] est en $O(N^3)$ avec N le nombre de travaux dans l'hyperpériode. Ils ont également proposé une heuristique pour réduire la complexité mais cette heuristique ne garantie pas l'optimalité de la taille de la période d'inactivité.

Le problème de cette approche est que même si la taille de la période d'inactivité courante est maximisée, plusieurs périodes d'inactivité de faible durée peuvent apparaître dans la suite de l'hyperpériode, et cette solution n'essaye en aucun cas de maximiser leur taille. Il est toujours possible de répéter l'algorithme pour calculer la taille optimale de la prochaine période d'inactivité mais cette solution n'optimise toujours que la période d'inactivité courante. Or, il peut être bénéfique de renoncer à une période d'inactivité pour obtenir des périodes d'inactivité plus larges par la suite.

Nous illustrons ce problème à l'aide de l'ensemble de tâches 3 du Tableau 3.3. Il est composé de 3 tâches périodiques caractérisées par leur WCET et leur période, son hyperpériode est de 12. Sur la Figure 3.6, deux périodes d'inactivité sont créées car l'algorithme d'ordonnancement

a essayé de créer la plus large période d'inactivité à $t = 1$ en retardant les exécutions de la tâche τ_3 autant que possible. Mais la deuxième période d'inactivité à $t = 7$ est seulement de taille unitaire ce qui pourrait ne pas permettre l'activation d'un état basse-consommation suivant la valeur du BET. Au contraire, sur la Figure 3.7, la période d'inactivité n'a pas commencée $t = 1$ mais à $t = 2.5$ pour permettre la création de deux périodes d'inactivité de taille identique. La consommation énergétique sera donc inférieure pour la Figure 3.7 car les deux périodes d'inactivité permettent l'activation d'un état basse-consommation.

	τ_1	τ_2	τ_3
WCET	1	2	3
Période	1.5	3	12

Table 3.3 – Ensemble de tâches 3.

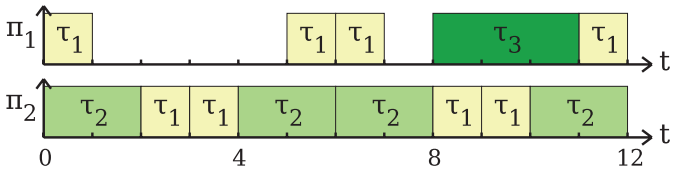

Figure 3.6 – Ordonnancement de l'ensemble de tâches 3 minimisant la taille de la prochaine période d'inactivité.

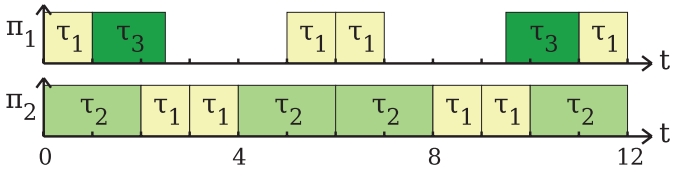

Figure 3.7 – Ordonnancement de l'ensemble de tâches 3 minimisant la taille de toutes les périodes d'inactivité sur l'hyperpériode.

3.3 Cadres d'application

Une fois l'algorithme défini pour optimiser la taille des périodes d'inactivité, les objectifs sont différents suivant le cadre d'application. Suivant la modélisation du système et son comportement lors de l'exécution, il est en effet possible d'optimiser davantage la taille des périodes d'inactivité à l'exécution. Dans cette section nous nous intéressons d'abord aux systèmes temps réel dur puis aux systèmes temps réel à criticité mixte.

3.3.1 Systèmes temps réel dur

Lors de l'exécution, chaque travail est susceptible de n'utiliser qu'une partie de son WCET. Comme vu au chapitre 2, nous appelons le temps processeur non utilisé par une tâche le *slack time* pour rester cohérent avec la littérature existante.

Par conséquent, l'utilisation des processeurs diminue en-ligne et la consommation énergétique également car il est possible d'activer des états basse-consommation pendant des périodes de temps plus importantes. L'algorithme d'ordonnancement doit alors utiliser en-ligne ce *slack time* pour réduire la consommation énergétique en augmentant la taille des périodes d'inactivité. Ces décisions ne peuvent se prendre qu'en-ligne car le temps d'exécution réel de chaque travail (i.e. AET) ne peut être connu préalablement à l'exécution. On retrouve alors tous les problèmes inhérents à ces décisions, comme par exemple la complexité pour retarder l'exécution des tâches afin de maximiser la taille d'une ou de plusieurs périodes d'inactivité. Plusieurs solutions ont été proposées [BBA10c, AY03, NG09], mais principalement pour exploiter ce *slack time* dans le but de réduire la consommation dynamique en diminuant la fréquence du processeur.

Une solution « naïve » serait d'utiliser le temps processeur libéré pour immédiatement activer un état basse-consommation et reprendre ensuite l'exécution des tâches comme prévu. Mais cette solution n'est pas efficace car la taille des périodes d'inactivité n'est alors pas optimisée et peut alors être inférieure au BET ce qui n'autorise pas l'activation d'un état basse-consommation. Cette solution ne permet également pas le regroupement de plusieurs périodes d'inactivité, ce qui est souhaitable pour activer des états basse-consommation plus économes en énergie.

3.3.2 Systèmes temps réel à criticité mixte

Les problèmes énoncés dans ce chapitre se concentrent pour l'instant sur des systèmes temps réel dur, c'est-à-dire des systèmes temps réel où aucun dépassement d'échéances n'est autorisé. Cependant, des modélisations alternatives sont possibles où certaines tâches peuvent tolérer des dépassements d'échéances, systèmes que nous avons présentés sous le nom de systèmes temps réel à criticité mixte. Les travaux énoncés ci-dessus restent valides pour ces systèmes mais ne permettent pas de tirer parti des systèmes temps réel à criticité mixte qui ont la particularité d'avoir des tâches à faible criticité. Il est possible de tirer parti de ces tâches qui, au contraire des tâches à haute criticité, peuvent tolérer selon leur criticité un certain nombre de dépassements d'échéances.

Comme vu ci-dessus, réduire la consommation statique nécessite d'avoir de larges périodes d'inactivité pour activer les états basse-consommation les plus efficaces énergétiquement. Pour créer de larges périodes d'inactivité avec l'ajout de tâches à faible criticité, il est possible de parier sur le fait que les tâches temps réel n'utilisent que rarement la totalité de leur WCET lors de l'exécution. Ainsi, tout en garantissant les échéances de toutes les tâches à haute criticité, il est possible d'être agressif sur les tâches à faible criticité en proposant un ordonnancement qui prévoit que ces tâches n'utiliseront qu'une partie de leur WCET. Il s'agit de trouver le meilleur compromis entre réduire la consommation statique et le respect des échéances des tâches à faible criticité. Le niveau de confiance sur la valeur du WCET est

différent selon le niveau de criticité de la tâche. Plus la tâche est critique, plus il est nécessaire d'être pessimiste et de s'assurer qu'aucune échéance ne sera violée si la tâche consomme la totalité de son WCET.

Lorsqu'une tâche est exécutée, il est probable qu'elle ne consomme pas la totalité de son WCET. Pour diminuer la consommation énergétique, l'algorithme d'ordonnancement peut ensuite s'appuyer sur ce point pour ordonnancer le système de telle sorte que les états basse-consommation les plus efficaces soient utilisés et donc que de larges périodes d'inactivité soient créées. La différence avec les systèmes périodiques est que l'algorithme d'ordonnancement a plus de possibilités qui lui sont offertes du fait de la présence de tâches à faible criticité.

3.4 Conclusion

Ce chapitre a montré que pour réduire la consommation énergétique en activant les états basse-consommation des processeurs, il est nécessaire d'optimiser la taille des périodes d'inactivité dans le but d'obtenir des périodes d'inactivité qui permette d'utiliser les états basse-consommation les plus efficaces. C'est-à-dire les états basse-consommation les plus économes en énergie tout en minimisant les délais de transition nécessaire pour revenir d'un état basse-consommation à l'état actif.

Optimiser la taille des périodes d'inactivité nécessite de retarder l'exécution des tâches, ce qui requiert que l'algorithme d'ordonnancement soit oisif et augmente la complexité de l'algorithme d'ordonnancement. Pour réduire la consommation énergétique globalement durant toute l'exécution du système, il est aussi nécessaire de prendre des décisions au niveau global et non local. En effet, optimiser uniquement la taille de la prochaine période d'inactivité peut être négatif pour la consommation énergétique pour la suite de l'exécution du système.

Les objectifs diffèrent également suivant la modélisation du système et les critères du concepteur. Nous avons vu qu'une modélisation optimiste autorisant certains dépassements d'échéances permet une plus grande agressivité sur l'optimisation des périodes d'inactivité. Au contraire, une modélisation pessimiste où aucun dépassement d'échéances n'est autorisé laisse moins de possibilité pour réduire la consommation énergétique.

Nous développons au prochain chapitre l'approche que nous proposons pour résoudre ce problème, à la fois pour les systèmes temps réel dur et les systèmes temps réel à criticité mixte. Les deux chapitres suivants détaillent ensuite les algorithmes d'ordonnancement proposés.

CHAPITRE 4

Approche hybride

Sommaire

Le chapitre précédent a présenté les différents problèmes liés à l'activation d'états basse-consommation pour réduire la consommation statique des systèmes temps réel multiprocesseurs. Dans ce chapitre, nous détaillons l'approche que nous allons suivre, à savoir son fonctionnement général, les hypothèses que nous faisons et la modélisation du temps d'inactivité du système que nous adoptons. La suite du chapitre traite des aspects en-ligne de l'approche proposée et de sa complexité. Les chapitres suivants détailleront les algorithmes d'ordonnancement complets utilisant cette approche.

4.1 Fonctionnement général

Nous avons vu dans le chapitre 2 qu'un algorithme d'ordonnancement est séparé en deux parties : hors-ligne et en-ligne. Un algorithme d'ordonnancement en-ligne fait la totalité de ses calculs lors de l'exécution tandis qu'un algorithme hors-ligne les effectue en majorité avant l'exécution. Une approche hors-ligne possède deux avantages principaux. Premièrement, la complexité et le temps de calcul ne sont pas problématiques. Ils seront de toute façon inférieurs au temps nécessaire pour concevoir et implémenter un système embarqué temps réel. Il est donc possible d'utiliser les solutions les plus performantes bien que plus complexes et qui seraient impossibles à mettre en œuvre en-ligne. Deuxièmement, la complexité en-ligne est réduite par

rapport à une solution en-ligne. En effet, il suffit lors de l'exécution de reproduire les décisions d'ordonnancement prises hors-ligne. En-ligne, lors de l'exécution du système, le principal facteur limitant est la complexité temporelle de l'algorithme d'ordonnancement. Comme vu au chapitre 2, l'hypothèse faite dans la grande majorité des travaux sur l'ordonnancement temps réel est l'instantanéité des décisions d'ordonnancement qui sont donc transparentes lors de l'exécution du système. Or cette hypothèse n'est correcte que si la complexité en-ligne de l'algorithme est limitée. Cette contrainte limite les possibilités d'un algorithme en-ligne, surtout dans notre situation étant donnée la complexité nécessaire pour générer de larges périodes d'inactivité dans un ordonnancement. Afin de tirer parti au mieux des avantages des deux approches, nous avons choisi de traiter le problème de manière hybride, c'est-à-dire avec une partie hors-ligne et une autre en-ligne.

Les calculs qui ne peuvent être traités hors-ligne sont les opérations dont les données en entrée ne peuvent être connues avant l'exécution. Pour notre problème, ce sont essentiellement les calculs qui requièrent la connaissance des temps d'exécution réels des tâches. En prenant en compte les WCET de toutes les tâches, notre approche calcule hors-ligne un ordonnancement sur une hyperpériode. En-ligne, il suffit à l'ordonnanceur d'avoir en mémoire les informations d'ordonnancement (i.e. quand et sur quel processeur exécuter une tâche) pour exécuter le système. La complexité est alors grandement réduite par rapport à une approche uniquement en-ligne.

4.1.1 Ordonnancement hors-ligne

Notre objectif est de prendre hors-ligne le maximum de décisions permettant d'obtenir un ordonnancement minimisant la consommation statique en utilisant les WCET des tâches. Générer un ordonnancement signifie prendre toutes les décisions d'ordonnancement nécessaires à l'exécution du système. Une fois ces informations connues, il suffit au système de les utiliser en-ligne. Le premier impératif de cet ordonnancement est de garantir les échéances de toutes les tâches du système. Une fois cette contrainte garantie, l'ordonnancement créé a pour second objectif de réduire la consommation statique.

Comme vu dans le chapitre précédent, les solutions d'ordonnancement partitionnées ne peuvent être aussi efficaces que les solutions globales pour optimiser la taille des périodes d'inactivité. En effet, une approche globale permet d'avoir une vision complète de l'ordonnancement et des périodes d'inactivité et donc de prendre de meilleures décisions pour optimiser la taille des périodes d'inactivité. Cette approche autorise la migration des tâches et des travaux entre processeurs durant l'exécution. Elle est également oisive pour permettre à un processeur d'être dans un état basse-consommation même si des tâches sont disponibles pour être exécutées.

Une approche hors-ligne est moins contrainte qu'une approche en-ligne. Elle n'est pas obligée de prendre ses décisions d'ordonnancement lorsqu'une tâche finit son exécution ou lorsqu'une tâche devient active comme le font les algorithmes d'ordonnancement multiprocesseurs classiques. Il est ainsi possible de découper l'exécution des tâches, de placer telle ou telle section d'une tâche sur tel ou tel processeur. Les décisions ne se prennent pas tâche par tâche mais d'un point de vue global sur la période de temps concernée, ici l'hyperpériode. Cette approche présente l'avantage d'être beaucoup plus flexible pour ordonnancer les tâches :

l'exécution, le retard de l'exécution d'une tâche ou la planification d'une préemption sont facilités. Nous utilisons ici cette flexibilité pour optimiser la taille des périodes d'inactivité afin d'activer les états basse-consommation les plus efficaces.

Hors-ligne, le calcul doit se faire en prenant en compte le WCET des tâches afin de garantir toutes les échéances. Comme les tâches peuvent terminer leur exécution avant d'avoir atteint leur WCET, le temps d'inactivité avec lequel est généré l'ordonnancement hors-ligne ne peut qu'augmenter lors de l'exécution. La consommation énergétique calculée hors-ligne sera par conséquent la limite supérieure de la consommation énergétique lors de l'exécution. Minimiser la consommation énergétique hors-ligne est alors bénéfique car cela permet de garantir une limite haute pour la consommation énergétique lors de l'exécution, ce que ne permet pas une approche en-ligne.

4.1.2 Hypothèses et contraintes

Dans cette partie nous détaillons et justifions les hypothèses prises pour générer notre algorithme d'ordonnancement.

4.1.2.1 Hyperpériode

Nous avons choisi d'adopter la réduction proposée au chapitre précédent en section 3.2.1, à savoir de restreindre l'intervalle d'étude à une hyperpériode.

4.1.2.2 Prise en compte de l'exécution du système

Comme la majorité des travaux sur l'ordonnancement temps réel, nous faisons l'hypothèse que le temps d'exécution de l'ordonnanceur lors de l'exécution du système peut être ignoré. Chaque exécution de l'ordonnanceur est considérée comme étant immédiate. L'argument utilisé pour justifier ce choix est de dire que ce temps d'exécution peut être compris dans les WCET des tâches car les tâches ne consomment pas tout leur WCET lors de l'exécution. Ces travaux ont également comme objectif de limiter la complexité de l'algorithme d'ordonnancement.

4.1.2.3 Utilisation globale

Pour réduire la consommation énergétique, la seule solution possible est d'utiliser le temps d'inactivité du système, soit la différence entre le nombre de processeurs m et l'utilisation globale U. Plus l'utilisation globale est proche du nombre de processeurs, plus le temps d'inactivité est limité et plus les possibilités d'utiliser les états basse-consommation seront réduites. Au contraire, plus l'utilisation globale du système est faible, plus les périodes d'inactivité peuvent être étendues.

Ainsi, si $U < m - 1$, il est possible d'obtenir un ordonnancement où au minimum un processeur pourra être placé dans un état basse-consommation durant la totalité de l'exécution sans qu'aucune échéance ne soit violée. Ce processeur est donc inutile dans la modélisation du système et peut donc être supprimé. En généralisant et en ne gardant dans notre modèle que les processeurs qui sont indispensables pour garantir l'ordonnançabilité du système, nous faisons l'hypothèse suivante pour le reste de ce chapitre :

$$m - 1 < U < m \tag{4.1}$$

Ainsi, tous les processeurs présents doivent être utilisés. Si d'autres processeurs sont disponibles, ils seront dans un état basse-consommation durant la totalité de l'exécution, ayant ainsi une consommation énergétique nulle.

4.1.2.4 Préemptions et migrations

Il est important que le nombre de préemptions et de migrations reste comparable à celui généré par les algorithmes multiprocesseurs optimaux existants les plus performants dans ce domaine car ces opérations peuvent avoir un coût énergétique non négligeable. Bastoni et al. [BBA10b, BBA10a] ont montré empiriquement que les préemptions et les migrations peuvent avoir des conséquences sur la consommation énergétique des systèmes. Leurs résultats montrent que les coûts des préemptions et des migrations dépendent de la taille des caches, principalement du cache L2.

Cependant, nous avons choisi de traiter ce point ultérieurement et d'essayer de résoudre ce problème seulement une fois l'ordonnancement généré. Nous montrerons au chapitre suivant que le nombre de préemptions et de migrations de notre solution est sensiblement égal à celui obtenu avec des algorithmes d'ordonnancement multiprocesseurs spécialement conçus pour optimiser ce critère. Nous proposons en section 7.2.2.4 une solution pour limiter le nombre de préemptions et migrations.

4.1.3 Modélisation hors-ligne de l'inactivité du processeur

Le modèle de tâches présenté au chapitre 2 ne permet pas d'avoir une influence sur les périodes d'inactivité autrement que par l'ordonnancement des tâches. Or, réduire la consommation statique nécessite de pouvoir manipuler les périodes d'inactivité du processeur, ce qui n'est pas possible avec la modélisation actuelle. Pour remédier à ce problème, nous avons ajouté dans le modèle de tâche initial une tâche périodique que nous nommons τ'. Les caractéristiques de cette tâche sont résumées dans le Tableau 4.1.

Période	H
Utilisation	$m - U$

Table 4.1 – Période et utilisation de la tâche τ'.

La période de la tâche τ' étant égale à l'hyperpériode, cette tâche n'a qu'un seul travail dans une hyperpériode. Pour que τ' représente réellement le temps d'inactivité, il faut que son utilisation corresponde à l'utilisation restante de l'ensemble de tâches, soit $m-U$. L'utilisation globale de l'ensemble de tâches avec τ' est donc maintenant égale au nombre de processeurs (i.e. m) soit de 100 %. Il ne reste plus de temps d'inactivité. La tâche τ' possède les même propriétés que les autres tâches de l'ensemble de tâches initial, à savoir par exemple qu'elle ne peut être exécutée simultanément sur deux processeurs. Cette modélisation n'est possible que

parce le temps d'inactivité du système est inférieur à l'hyperpériode du fait de l'hypothèse sur l'utilisation globale de l'équation (4.1).

Par conséquent, l'ajout de τ' dans l'ensemble de tâches n'est pas neutre d'un point de vue de l'ordonnancement et certains ordonnancements valides sans τ' ne le sont maintenant plus avec τ'. Par exemple, il n'est plus possible pour deux processeurs d'être inactifs simultanément car cela signifierait que τ' est en cours d'exécution sur ces processeurs. Mais notre objectif étant de réduire la consommation énergétique en activant les états basse-consommation, l'ajout de τ' n'est pas un problème car nous voulons optimiser la taille des périodes d'inactivité et également réduire leur nombre. Avec l'ajout de τ', un seul processeur active des états basse-consommation si τ' est toujours exécutée sur le même processeur.

La tâche τ' est utilisée pour optimiser hors-ligne la consommation statique et le système activera en-ligne un état basse-consommation si possible lors de l'exécution de cette tâche. Ajouter cette tâche dans l'ensemble de tâches permet de modéliser hors-ligne le temps d'inactivité du système. Mais lors de l'exécution du système, il est tout à fait possible qu'un processeur soit inactif en dehors de l'exécution de τ', par exemple lorsqu'une tâche ne consomme pas tout son WCET.

Notre modélisation est indépendante de l'hypothèse que nous avons fait de réduire l'intervalle d'étude à une hyperpériode. Il est cependant possible de travailler sur une durée plus longue si le concepteur dispose d'un puissance de calcul plus importante. Effectuer le calcul sur deux hyperpériodes permet également de prendre en compte une période d'inactivité entre ces deux hyperpériodes. La tâche τ' aurait alors une période T égale à $2 \times H$ (où H est l'hyperpériode) et un temps d'exécution de $2 \times (m - U)$.

4.2 Résolution par programmation linéaire

Nous exprimons dans cette section le problème d'obtenir un ordonnancement valide réduisant la consommation énergétique sous la forme d'un programme linéaire. La solution est un ordonnancement permettant de réduire la consommation statique.

Pour générer notre ordonnancement, nous utilisons l'approche proposée par Lemerre et al. [LDAVN08]. Cette approche est également utilisée par Mégel et al. [Mé12, MSD10] pour proposer un algorithme d'ordonnancement multiprocesseur optimal réduisant le nombre de préemptions et de migrations. Au lieu de représenter un ordonnancement de manière classique où des tâches sont assignées aux processeurs, cette approche exprime le problème d'ordonnancement comme le fait d'assigner un temps d'exécution à chaque tâche sur des intervalles. Les auteurs ont démontré que ces deux représentations sont équivalentes. Cette approche ne fait pas partie de notre contribution mais nous la présentons dans ce chapitre dans un souci d'homogénéité.

Cette approche est globale et ne restreint pas les migrations et les préemptions des tâches. Chaque tâche peut être préemptée et peut changer de processeur à chaque instant de l'exécution. Elle n'utilise pas les priorités pour ordonnancer les tâches : une tâche peut être ordonnancée avant une autre à un instant t et après cette même autre tâche à un autre moment de l'exécution.

Cette technique permet de construire un algorithme d'ordonnancement multiprocesseur

optimal capable d'ordonnancer tous les ensembles de tâches dont l'utilisation globale est inférieure ou égale au nombre de processeurs. Contrairement à l'algorithme PFair [BCPV93], cet algorithme ne divise pas l'exécution en « quantum » de temps. De la même façon que BFair [ZMM03], l'exécution est divisée en intervalles où les frontières des intervalles sont les périodes des tâches.

Nous utilisons cette approche car elle permet d'exprimer formellement l'ensemble du problème y compris les contraintes et les objectifs. Un des défauts des solutions actuelles est le fait que ces solutions ont pour objectif principal le respect de toutes les échéances et l'objectif secondaire (i.e. réduire la consommation énergétique) est difficile à atteindre vu la complexité du premier objectif. Au contraire, en utilisant cette approche, respecter les échéances est vu comme une contrainte et il est aisé d'exprimer notre objectif de réduire la consommation énergétique.

Dans le suite de cette section, nous présentons la modélisation propre à cette approche, les contraintes relatives aux échéances des tâches et enfin comment les tâches sont ordonnancées à l'intérieur des intervalles. Les différentes fonctions objectifs que nous proposons seront ensuite détaillées en section 4.3.

4.2.1 Division de l'hyperpériode en intervalles

Dans cette représentation, l'exécution du système est divisée en intervalles, les frontières des intervalles étant les dates de demande d'activation des travaux. Nous nommons I l'ensemble des intervalles et $|I_k|$ la taille de l'intervalle k. Le découpage en intervalle se fait sur toute la durée de l'hyperpériode.

Sur chaque intervalle, le poids d'une tâche représente la portion de processeur qui est réservée à cette tâche sur cet intervalle. Nous notons $w_{j,k}$ la variable de décision correspondant au poids du travail j sur l'intervalle k avec $w_{j,k}$ un nombre réel $\in [0, 1]$. Le temps d'exécution du travail j sur l'intervalle k sera ensuite égale à $w_{j,k} \times |I_k|$.

Dans cette représentation, chaque travail a un identifiant qui lui est propre, deux travaux de deux tâches différentes ne peuvent avoir le même identifiant. J_k est le sous-ensemble de l'ensemble des travaux J_Γ qui contient tous les travaux présents sur l'intervalle k. E_j est l'ensemble des intervalles sur lesquels le travail j est présent. E_j contient au moins un intervalle. Ces notations sont résumées dans le Tableau 4.2.

I	Ensemble des intervalles
$\|I_k\|$	Taille de l'intervalle k
$w_{j,k}$	Poids du travail j dans l'intervalle k
$w_{j,k} \times \|I_k\|$	Temps d'exécution du travail j dans l'intervalle k
J_k	Ensemble des travaux présents sur l'intervalle k
E_j	Ensemble des intervalles sur lesquels le travail j est présent

Table 4.2 – Définitions des intervalles et des poids associés aux tâches.

Exemple. L'ensemble de tâches 4 du Tableau 4.3 est composé de trois tâches et son hyperpériode est de 12. La Figure 4.1 représente les travaux et les intervalles de cet ensemble de tâches sur une hyperpériode. Les tâches τ_1 et τ_2 ont chacune trois travaux tandis que τ_3 en a deux. Sur la Figure 4.2 sont représentés tous les poids sur une hyperpériode. Le travail j_2 étant présents sur les deux intervalles I_2 et I_3, nous avons $E_2 = \{2, 3\}$ et son temps d'exécution total est de $w_{2,2} \times |I_2| + w_{2,3} \times |I_3|$. Sur cet exemple, $J_1 = \{1, 4, 7\}$ et $J_2 = \{2, 5, 7\}$.

	τ_1	τ_2	τ_3
WCET	0.8	2.4	4
Période	4	4	6

Table 4.3 – Ensemble de tâches 4.

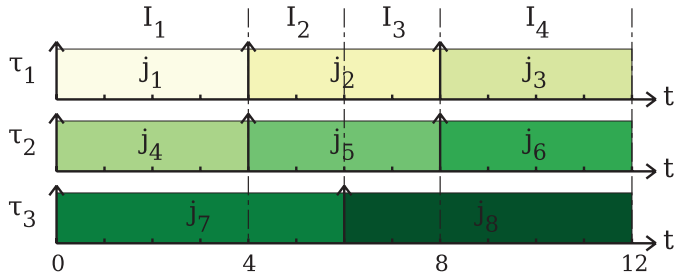

Figure 4.1 – Travaux et intervalles de l'ensemble de tâches 4.

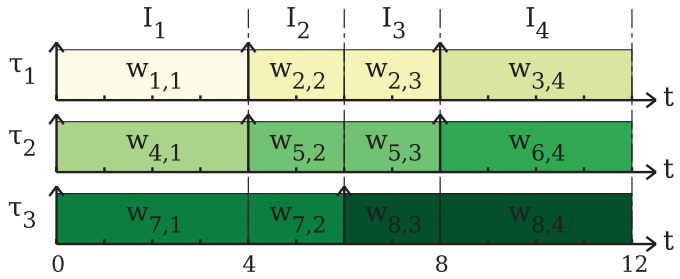

Figure 4.2 – Poids des travaux de l'ensemble de tâches 4.

Nous discutons maintenant de la complexité de cette approche. La suite de cette section présente ensuite les contraintes du programme linéaire et l'ordonnancement des tâches à l'intérieur des intervalles.

Complexité du programme linéaire. L'approche utilisée a pour objectif de déplacer au maximum la complexité de l'ordonnancement hors-ligne pour que la complexité en-ligne (spatiale ou temporelle) soit minimale.

La complexité du programme linéaire dépend du nombre d'intervalles dans l'hyperpériode qui est lié au nombre de travaux J dans l'hyperpériode, J ayant déjà été défini au chapitre 2. Dans le pire cas, chaque travail crée un intervalle sauf tous les derniers travaux de chaque tâche dont l'échéance est la fin de l'hyperpériode. Ce pire cas intervient quand toutes les périodes sont premières entre elles. Si I est le nombre d'intervalles dans une hyperpériode, I est donc borné pour n tâches par :

$$I \leq J - n + 1 \tag{4.2}$$

Ce nombre peut en théorie être très élevé mais reste raisonnable en pratique. Les systèmes temps réel existants ont en effet des tâches dont les périodes sont des harmoniques, c'est-à-dire que les valeurs des périodes ont des diviseurs en commun. Un ensemble de tâches dont les périodes sont importantes et toutes premières entre elles est peu probable.

Complexité spatiale. Pour chaque tâche, le poids de chaque travail sur chaque intervalle doit être stocké dans la mémoire du système pour pouvoir être utilisé lors de l'exécution. Les temps d'exécution peuvent à la place être stockés (i.e. $w_{j,k} \times |I_k|$), mais la complexité est identique. Ce qui représente $n \times I$ valeurs à stocker, avec n le nombre de tâches et I le nombre d'intervalles. Cependant, si cette matrice est suffisamment vide, il peut être plus efficace de ne stocker que les poids avec leur intervalle et leur tâche associée. Nous définissons e comme le nombre de valeurs non nulles de la matrice. $e \times 3$ entiers doivent en conséquent être gardés en mémoire.

De plus, les priorités de chaque tâche peuvent également être calculées hors-ligne pour limiter la complexité en-ligne, ce qui représente $n \times I$ valeurs supplémentaire. La place nécessaire en mémoire est donc de :

$$min(n \times I, e \times 3) + n \times I \tag{4.3}$$

Le facteur important est donc le nombre d'intervalles dans une hyperpériode. Nous avons discuté de ce problème au chapitre 2, la conclusion étant que le nombre d'intervalles doit être limité pour d'autres raisons que la complexité de l'algorithme d'ordonnancement. Nous pensons donc que la complexité spatiale de notre solution ne pose pas de problèmes d'un point de vue pratique.

Comme il est nécessaire de sauvegarder le poids de chaque tâche dans chaque intervalle, le nombre de valeurs à conserver en mémoire est donc égal à :

$$n \times I \tag{4.4}$$

Au lieu de conserver le poids de la tâche en mémoire, il peut être plus efficace de sauvegarder directement son temps d'exécution, soit le poids multiplié par la taille de l'intervalle. Il est également envisageable de calculer directement les priorités des tâches hors-ligne et de les conserver en mémoire. Comme pour les poids, il faut alors conserver la priorité de chaque tâche sur chaque intervalle. Le nombre d'entiers à conserver en mémoire devient donc :

$$n \times 2 \times I \tag{4.5}$$

Complexité en-ligne. La complexité en-ligne est la complexité nécessaire pour prendre les décisions d'ordonnancement lors de l'exécution. Notre solution limite cette complexité temporelle en-ligne en maximisant les calculs hors-ligne. Deux types de décisions doivent être prises lors de l'exécution, au début de chaque intervalle et à l'intérieur des intervalles. Nous discutons plus longuement de cette complexité en section 4.4 où est présenté en détail l'algorithme d'ordonnancement que nous utilisons en-ligne pour ordonnancer les travaux à l'intérieur des intervalles.

4.2.2 Contraintes d'ordonnançabilité

Un ensemble de contraintes est nécessaire pour garantir qu'aucune échéance n'est violée et que l'ensemble de tâches soit ordonnançable. Ces contraintes sont basées sur les travaux de Lemerre et al. [LDAVN08].

L'inéquation (4.6) garantit que l'utilisation globale de chaque intervalle ne dépasse pas le nombre de processeurs m, un intervalle avec une utilisation plus grande que m n'étant pas ordonnançable.

$$\forall k, \sum_{j \in J_k} w_{j,k} \leq m \tag{4.6}$$

L'inéquation suivante (4.7) contraint le poids d'un travail dans un intervalle entre 0 et 1 pour permettre l'exécution du travail en question. En effet, un travail avec un poids supérieur à 1 ne pourrait être ordonnancé car son temps d'exécution serait supérieur à la taille de l'intervalle et un travail ne peut être exécuté sur plus d'un processeur simultanément.

$$\forall k, \forall j,\ 0 \leq w_{j,k} \leq 1 \tag{4.7}$$

Enfin, l'équation (4.8) garantit que la somme des poids assignés à chaque travail est égal à son WCET. Cette équation est nécessaire pour garantir qu'aucune échéance n'est violée lors de l'exécution.

$$\forall j, \sum_{k \in E_j} w_{j,k} \times |I_k| = j.c \tag{4.8}$$

Ces trois équations (4.6), (4.7) et (4.8) forment le programme linéaire (PL 1). Résoudre ce programme permet d'obtenir le poids de chaque tâche sur chaque intervalle et donc un ordonnancement valide. Cependant, cet ordonnancement n'est pas optimisé pour obtenir de larges périodes d'inactivité ou pour réduire la consommation statique. Une fonction objectif doit pour cela être ajoutée, nous en discutons dans la section suivante en fonction du cadre d'application, à savoir les systèmes temps réel dur ou les systèmes temps réel à criticité mixte.

$$(PL\ 1) \qquad \begin{cases} \forall k, \sum_{j \in J_k} w_{j,k} \leq m \\ \forall k, \forall j,\ 0 \leq w_{j,k} \leq 1 \\ \forall j, \sum_{k \in E_j} w_{j,k} \times |I_k| = j.c \end{cases}$$

4.3 Cadres d'application

Dans cette section nous détaillons les spécificités de chaque cadre d'application. Pour les systèmes temps réel dur nous proposons deux fonctions objectifs différentes pour minimiser la consommation statique en garantissant les échéances de toutes les tâches. Pour les systèmes temps réel à criticité mixte, nous proposons une fonction objectif plus agressive tenant compte du fait que les tâches à faible criticité peuvent tolérer des dépassements d'échéances.

4.3.1 Systèmes temps réel dur

Dans le cadre des systèmes temps réel dur où aucun dépassement d'échéances n'est autorisé, nous proposons deux solutions différentes qui sont ensuite développées au chapitre suivant. Cette section donne l'idée générale de chaque solution.

La première solution se base sur une simplification dans la mesure où son objectif est de minimiser le nombre de périodes d'inactivité et non de minimiser la consommation énergétique en elle-même. En effet, comme les tâches utilisent leur WCET, le temps d'inactivité total est connu et minimiser le nombre de périodes d'inactivité permet de maximiser la taille de ces périodes d'inactivité. L'ordonnancement obtenu a donc un nombre minimal de périodes d'inactivité pour pouvoir activer les états basse-consommation les plus efficaces.

Comme vu au chapitre précédent en section 3.2.2, cette première solution ne tire pas parti des caractéristiques des processeurs. Selon le nombre d'états basse-consommation, leur consommation énergétique et leur pénalité, il est possible d'envisager des ensembles de tâches où minimiser le nombre de périodes d'inactivité n'est pas la solution la plus adaptée. Pour remédier à ce problème, notre seconde solution cherche à minimiser la consommation statique en calculant pour chaque période d'inactivité sa consommation énergétique. Ce qui permet de générer des périodes d'inactivité adaptées aux états basse-consommation du processeur utilisé.

4.3.2 Systèmes temps réel à criticité mixtes

Comme vu aux chapitres 2 et 3, un ensemble de tâches à criticité mixte est un ensemble de tâches périodique où chaque tâche est également caractérisée par son niveau de criticité. Nous n'utiliserons dans le cadre de cette thèse que deux niveaux de criticité : *HIGH* et *LOW*. Les tâches ayant une criticité *HIGH* sont des tâches temps réel dur où aucun dépassement d'échéances n'est autorisé tandis que les tâches dont la criticité est *LOW* peuvent tolérer des dépassements d'échéances. Nous proposerons au chapitre 7 des pistes pour prendre en charge davantage de niveaux de criticité.

L'approche que nous utilisons pour les systèmes temps réel à criticité mixte s'appuie sur celle qui a été proposée ci-dessus pour les systèmes temps réel dur. Hors-ligne, un programme linéaire est utilisé pour réserver le temps d'exécution de chaque tâche. Cependant, contrairement aux systèmes temps réel dur, nous ne réservons un temps d'exécution égal au WCET que pour les tâches à haute criticité.

Notre approche est d'ordonnancer les tâches à faible criticité de façon plus agressive en ne réservant qu'un pourcentage de leur WCET hors-ligne. Ne réserver qu'une partie du WCET

des tâches à faible criticité permet de réduire le temps d'exécution prévu pour ces tâches et donc d'avoir plus de temps d'inactivité pour utiliser les états basse-consommation et réduire la consommation énergétique. L'inconvénient de cette approche est que les tâches à faible criticité sont susceptibles de manquer leurs échéances suivant la valeur de leur AET lors de l'exécution. Cependant, les tâches n'utilisent que rarement leur WCET lors de l'exécution et pourront donc néanmoins terminer leur exécution suivant la valeur de leur AET. Pour allouer au mieux les temps d'exécution des tâches à faible criticité nous utilisons les lois de distribution des temps d'exécution réels de ces tâches.

Le WCET d'une tâche est une valeur donnée et difficilement calculable. Plus le niveau de criticité d'une tâche est élevé plus le temps d'exécution alloué est important par rapport au temps d'exécution réel de la tâche. Cette approche permet de réduire cette différence pour les tâches à faible criticité tout en pariant en-ligne que d'autres tâches fourniront éventuellement du temps processeur en terminant leur exécution avant leur WCET.

4.4 Ordonnancement en-ligne

Une fois les poids de toutes les tâches connus grâce à la résolution du programme linéaire, le système peut s'exécuter. Lors de l'exécution de τ', l'état basse-consommation le plus adapté à la taille de la période d'inactivité courante peut être activé. Comme le poids de toutes les tâches est connu, la taille de la période d'inactivité peut être calculée au début de cette période d'inactivité.

Cette section est valable pour les systèmes temps réel dur et les systèmes temps réel à criticité mixte, les spécificités de chaque approche seront traitées dans les chapitres suivants. À l'intérieur de chaque intervalle, le temps d'exécution de chaque tâche est connu après la résolution du programme linéaire. L'utilisation globale de chaque intervalle étant égale au nombre de processeurs, il est nécessaire d'utiliser un algorithme d'ordonnancement optimal pour ordonnancer les intervalles. Comme présenté au chapitre 2, il existe un nombre réduit d'algorithmes d'ordonnancement multiprocesseurs, et certains sont inutilisables en pratique à cause de leur complexité (e.g. U-EDF) ou du grand nombre de préemptions qu'ils engendrent (e.g. PFair). Plusieurs algorithmes d'ordonnancement multiprocesseurs suivent une autre approche utilisant la laxité des tâches, par exemple EDZL [WCL+07] ou IZL [MSD10].

FPZL. L'algorithme que nous utilisons est *Fixed Priority until Zero Laxity (FPZL)* [DB11a], qui est un algorithme d'ordonnancement à priorités fixes. Cet algorithme ordonnance les tâches suivant leur priorité et contrôle la laxité des tâches restantes pour exécuter une tâche dont la laxité devient nulle. Nous avons sélectionné FPZL car utiliser des priorités fixes permet d'ordonnancer facilement la tâche τ' différemment des autres tâches du système. τ' étant la tâche représentant le temps d'inactivité du système, nous souhaitons ordonnancer τ' à l'intérieur des intervalles de telle sorte que la consommation statique soit minimale.

Pour attribuer des priorités, nous nous basons sur le temps d'exécution des tâches à l'intérieur des intervalles. Plus le temps d'exécution est important, plus la priorité est élevée. La priorité de τ' est décidée indépendamment des autres tâches. En effet, seul l'ordonnancement de τ' nous importe pour réduire la consommation énergétique, l'ordonnancement des autres

tâches n'a lui aucun impact, du moins tant que les tâches utilisent leur WCET. L'attribution des priorités aux tâches autres que τ' est donc un problème secondaire et attribuer les priorités de manière différente est envisageable selon les besoins du système.

Nous utilisons FPZL uniquement pour ordonnancer l'intérieur des intervalles. Toutes les tâches ont alors la même échéance qui est égale à la fin de l'intervalle. Cette singularité simplifie la mise en œuvre de FPZL car il n'est plus nécessaire de contrôler la laxité de tous les travaux mais seulement du travail dont la laxité est la plus faible (i.e. le travail dont le temps d'exécution restant dans l'intervalle est le plus important).

Complexité. La complexité en-ligne de notre algorithme utilisation FPZL est proche de celle de l'algorithme d'ordonnancement IZL [MSD10]. Cependant, nous avons en plus la tâche τ' qui doit être ordonnancée de telle sorte que la taille des périodes d'inactivité soit optimisée lorsque $0 < w_k < 1$.

Au début de chaque intervalle, le système choisit la tâche à ordonnancer en utilisant les priorités. Choisir la tâche à exécuter nécessite de choisir m tâches parmi n à exécuter sur les m processeurs du système. Comme les tâches sont déjà rangées par priorités, cette décision se fait en $O(m)$.

A l'intérieur des intervalles, les décisions d'ordonnancement se prennent en $O(1)$ car il suffit de prendre la tâche suivante par ordre de priorité. A noter qu'il n'est pas nécessaire de surveiller la laxité de chaque tâche mais de la tâche avec la priorité la plus importante pour être sûr que les tâches de laxité nulle sont ordonnancées.

Optimalité de FPZL. FPZL n'est pas un algorithme d'ordonnancement multiprocesseur optimal mais il est optimal lorsque toutes les tâches partagent la même échéance. Cette optimalité n'avait à notre connaissance jamais été démontrée. Le théorème 1 prouve cette optimalité :

Théorème 1. *FPZL est un algorithme d'ordonnancement optimal lorsque les tâches partagent la même échéance.*

Démonstration. Supposons qu'une tâche τ n'ait pas terminé son exécution à la fin de l'intervalle, son échéance est donc violée. Cela signifie que la tâche τ avait une laxité négative lorsque son exécution a débutée. Donc τ n'avait pas été ordonnancée lorsque sa laxité était devenue nulle, ce qui ne peut se produire que si toutes les tâches en cours d'exécution ont également une laxité de zéro. Mais si toutes les tâches ont une laxité de zéro, toutes les tâches vont être exécutées jusqu'à la fin de l'intervalle alors que la tâche τ doit encore être exécutée. Or aucune période d'inactivité n'a pu être créée depuis le début de l'intervalle parce qu'au minimum m tâches ont encore du temps d'exécution restant. Cela signifie donc que l'utilisation globale de l'intervalle est supérieure à m, ce qui est en contradiction avec l'hypothèse que l'utilisation globale est comprise entre 0 et m. Il ne peut donc y avoir de dépassements d'échéances. □

Chaque algorithme ordonnance la tâche τ' de manière différente, tout en utilisant FPZL pour ordonnancer les intervalles. Ces spécificités seront abordées aux chapitre 5 et 6 lors de la présentation de chaque algorithme.

Tâches n'utilisant pas leur WCET. Lors de l'exécution du système, les tâches n'utilisent pas leur WCET mais un temps d'exécution réel qui lui est inférieur, que nous avons nommé AET au chapitre 2. Les périodes d'inactivité créées par l'approche hors-ligne ne peuvent voir leur taille se réduire lors de l'exécution mais elles peuvent par contre être allongées pour réduire davantage la consommation énergétique du système. Ce temps processeur est généralement nommé dans la littérature *slack time*. Le *slack time* peut être utilisé de différentes manières selon les besoins. Les différentes solutions proposées sont détaillées dans les chapitres suivants, selon que l'on veuille réduire la consommation énergétique, la complexité de l'algorithme voire le nombre de dépassements d'échéances.

4.5 Conclusion

Ce chapitre a détaillé l'approche adoptée pour résoudre les différents problèmes énoncés au chapitre précédent pour réduire la consommation statique des systèmes temps réel multiprocesseurs en optimisant la taille des périodes d'inactivité et l'utilisation des états basse-consommation.

L'approche que nous proposons est différente des solutions existantes du fait que la majorité des calculs sont effectués hors-ligne. En effet, réduire la consommation statique, donc optimiser la taille des périodes d'inactivité pour utiliser des états basse-consommation efficaces est une opération complexe. Nous pensons alors qu'effectuer cette opération hors-ligne est nécessaire. Les seuls calculs en-ligne sont ceux qui doivent être effectués lorsque les tâches terminent leur exécution avant leur WCET pour pouvoir de nouveau augmenter la taille des périodes d'inactivité.

Nous divisons l'hyperpériode en intervalles et utilisons la programmation linéaire pour allouer un temps d'exécution pour chaque tâche sur chaque intervalle. Pour modéliser le temps d'inactivité des processeurs, nous avons introduit une nouvelle tâche τ' dans l'ensemble de tâches de telle sorte que l'utilisation globale soit maintenant égale au nombre de processeurs. Un ensemble de contraintes permet de s'assurer du respect de toutes les échéances. Ensuite, suivant le cadre applicatif visé, nous proposons différente objectifs pour réduire la consommation énergétique.

Ces objectifs sont détaillés dans les deux prochains chapitres qui présentent LPDPM et LPDPM-MC, les deux algorithmes d'ordonnancements multiprocesseurs optimaux utilisant cette approche. Ils sont respectivement destinés aux systèmes temps réel dur et aux systèmes temps réel à criticité mixte.

CHAPITRE 5

Systèmes temps réel dur

Sommaire

Nous proposons dans ce chapitre deux algorithmes d'ordonnancement multiprocesseurs pour réduire la consommation énergétique des systèmes temps réel dur où aucun dépassement d'échéances n'est autorisé. Ces deux solutions sont basées sur la même modélisation et la même approche du chapitre précédent et sont donc regroupées en un seul chapitre. Elles utilisent pour cela deux fonctions d'optimisation du programme linéaire différentes. La première solution vise à minimiser le nombre de périodes d'inactivité. La seconde cherche à minimiser la consommation énergétique. Les deux algorithmes d'ordonnancement complets sont respectivement nommés LPDPM1 et LPDPM2 (pour *Linear Programming Dynamic Power Management*). Ils diffèrent également dans leur ordonnancement en-ligne à l'intérieur des intervalles. Ces deux algorithmes sont détaillés dans les deux premières sections puis évalués en fin de chapitre.

5.1 Minimisation du nombre de périodes d'inactivité

Cette section présente LPDPM1, un algorithme d'ordonnancement temps réel multiprocesseur optimal permettant de réduire la consommation énergétique en minimisant le nombre de périodes d'inactivité. LPDPM1 est un algorithme d'ordonnancement hybride utilisant l'approche du chapitre 4 avec deux parties hors et en-ligne. La partie hors-ligne découpe l'hyperpériode en intervalles et calcule en utilisant la programmation linéaire le

temps d'exécution de chaque tâche sur chaque intervalle de telle sorte que le nombre de périodes d'inactivité soit minimisé. La partie en-ligne cherche ensuite à ordonnancer les tâches à l'intérieur des intervalles et augmente la taille des périodes d'inactivité lorsque les tâches ne consomment pas tout leur temps d'exécution.

Ce premier algorithme d'ordonnancement est conçu de manière à réduire la complexité du programme linéaire. Ainsi, l'objectif, au lieu de minimiser la consommation statique, consiste à minimiser le nombre de périodes d'inactivité, ce qui représente une simplification par rapport au problème original comme vu au chapitre 3 en section 3.2.2. En supposant que chaque tâche utilise son WCET, le temps total d'inactivité sur une hyperpériode est connu. Minimiser le nombre de périodes d'inactivité est équivalent à maximiser leur taille moyenne, permettant ainsi l'utilisation d'états basse-consommation plus efficaces en énergie. Cette solution est une simplification car elle ne tire pas bénéfice de la connaissance des états basse-consommation de chaque processeur. Mais la solution donnée par LPDPM1 est valide sur tous les processeurs.

Comme vu au chapitre 4, un ordonnancement hors-ligne est obtenu en calculant le poids (i.e. $w_{j,k}$) de chaque travail sur chaque intervalle en utilisant la programmation linéaire. Nous avons vu au chapitre précédent les contraintes nécessaires pour garantir l'ordonnançabilité du système, nous ajoutons dans ce chapitre la fonction objectif pour minimiser le nombre de périodes d'inactivité.

L'inactivité du processeur est représentée par l'exécution de la tâche τ'. Chaque exécution de τ' représente donc une période d'inactivité et chaque préemption de τ' représente la fin d'une période d'inactivité. Pour réduire le nombre de périodes d'inactivité, nous souhaitons par conséquent réduire le nombre de préemptions de τ'.

Cet objectif est traité en deux parties. La première étape est de minimiser le nombre de préemptions à l'intérieur des intervalles tandis que la seconde consiste à réduire les préemptions entre deux intervalles successifs. Concrètement, cela revient tout d'abord à minimiser le nombre d'intervalles pour lesquelles $0 < w_k < 1$ car dans ces intervalles l'exécution de τ' entraîne nécessairement une préemption. Ensuite, l'objectif est de rendre intervalles où $w_k = 0$ consécutifs, de même pour les intervalles où $w_k = 1$.

5.1.1 Minimisation des préemptions à l'intérieur des intervalles

Pour illustrer notre cheminement et la progression de notre solution, nous utilisons l'ensemble de tâches 4 dont nous rappelons les caractéristiques dans le Tableau 5.1. L'hyper-période de cet ensemble de tâches est de 12 et elle est divisée en quatre intervalles dont la taille est respectivement de 4, 2, 2 et 4. Sans aucune fonction objectif, un ordonnancement possible de cet ensemble de tâches est présenté en Figure 5.1. Sur cette figure, quatre exécutions de τ' sont générées sur l'hyperpériode donc quatre périodes d'inactivité, une sur chaque intervalle. Nous avons donc $0 < w_k < 1$ sur chaque intervalle, ce que nous souhaitons éviter.

Pour représenter les intervalles où $0 < w_k < 1$ et minimiser le nombre de préemptions de τ' à l'intérieur des intervalles, nous définissons deux variables binaires f_k et e_k. Ces deux variables représentent respectivement les intervalles où τ' occupe l'intervalle entier (i.e. *full*, $w_k = 1$) et les intervalles où τ' n'est pas exécutée (i.e. *empty*, $w_k = 0$). Ainsi, f_k est tel que :

	τ_1	τ_2	τ_3
WCET	0.8	2.4	4
Période	4	4	6

TABLE 5.1 – Ensemble de tâches 4.

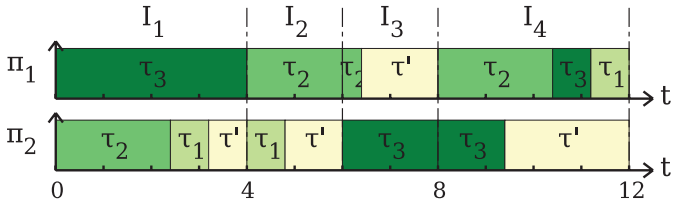

FIGURE 5.1 – Ordonnancement de l'ensemble de tâches 4 sans fonction objectif.

$$f_k = \begin{cases} 0 & \text{si } w_k = 1 \\ 1 & \text{sinon} \end{cases} \tag{5.1}$$

La somme des f_k de tous les intervalles de l'hyperpériode représente donc le nombre d'intervalles où le poids de τ' est différent de 1. Pour minimiser ce nombre et minimiser le nombre d'intervalles où $w_k \neq 1$, la fonction objectif est :

$$\text{Minimiser} \sum_k f_k \tag{5.2}$$

De manière similaire pour les intervalles où le poids de τ' est différent de 0, e_k est tel que :

$$e_k = \begin{cases} 0 & \text{si } w_k = 0 \\ 1 & \text{sinon} \end{cases} \tag{5.3}$$

Le programme linéaire en variables mixtes composé des équations (4.6), (4.7), (4.8), (5.1) et (5.3) et ayant pour objectif de minimiser $\sum_k f_k + e_k$ permet donc de générer un ordonnancement où le nombre d'intervalles pour lesquels $0 < w_k < 1$ est minimisé. Un ordonnancement possible avec cette fonction objectif est représenté en Figure 5.2. Le nombre d'exécutions de τ' sur une hyperpériode est passé de quatre à trois en comparaison avec la situation de la Figure 5.1. Le poids de τ' est maintenant nul sur l'intervalle 2 et égal à un sur l'intervalle 3.

5.1.2 Minimisation des préemptions entre deux intervalles consécutifs

Pour minimiser le nombre de préemptions entre deux intervalles consécutifs, nous voulons que deux intervalles où $f_k = 0$ soient consécutifs, de même pour les intervalles où $e_k = 0$. Nous avons choisi pour cela d'ajouter deux variables binaires fc_k et ec_k afin de représenter les transitions entre les intervalles. fc_k est égale à 1 lorsque $f_k = 1$ et $f_{k+1} = 0$. Soit :

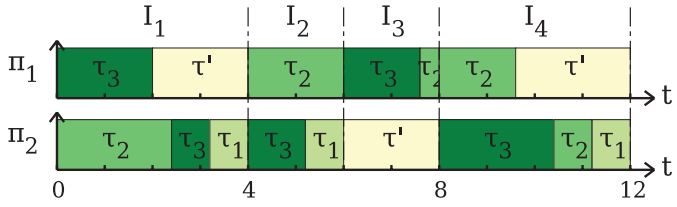

FIGURE 5.2 – Ordonnancement de l'ensemble de tâches 4 avec comme fonction objectif $Min \sum_k f_k + e_k$.

$$fc_k = \begin{cases} 1 & \text{si } f_k = 1 \text{ et } f_{k+1} = 0 \\ 0 & \text{sinon} \end{cases} \tag{5.4}$$

La variable binaire fc_k permet de différencier les transitions entre intervalles où la valeur de f_k passe de 1 à 0, soit les transitions entre deux intervalles où le poids de τ' est différent de 1 puis ensuite égal à 1. Réduire ces transitions permet de rendre les intervalles où $w_k = 1$ consécutifs, de même pour les intervalles où $w_k \neq 1$. Cette équation (5.4) peut être linéarisée ainsi :

$$\begin{cases} f_k - f_{k+1} - fc_k \leq 0 \\ -f_k + fc_k \leq 0 \\ f_{k+1} + fc_k \leq 1 \end{cases} \tag{5.5}$$

Nous utilisons un raisonnement identique pour les intervalles où $e_k = 0$, soit ec_k tel que :

$$ec_k = \begin{cases} 1 & \text{si } e_k = 1 \text{ et } e_{k+1} = 0 \\ 0 & \text{sinon} \end{cases} \tag{5.6}$$

De même, cette équation peut être linéarisée ainsi :

$$\begin{cases} e_k - e_{k+1} - ec_k \leq 0 \\ -e_k + ec_k \leq 0 \\ e_{k+1} + ec_k \leq 1 \end{cases} \tag{5.7}$$

Pour minimiser le nombre de préemptions entre deux intervalles consécutifs, la fonction objectif est :

$$\text{Minimiser} \sum_k fc_k + ec_k \tag{5.8}$$

5.1.3 Fonction d'optimisation

Toutes les variables nécessaires à la construction de notre fonction objectif ont maintenant été introduites. Ainsi, le programme linéaire en variables mixtes est composé des équations (4.6), (4.7), (4.8), (5.1), (5.3), (5.5) et (5.7). Et sa fonction objectif est :

$$\text{Minimiser} \sum_k f_k + e_k + fc_k + ec_k \tag{5.9}$$

L'ajout de cette fonction objectif finalise notre programme linéaire (PL 2). Le résoudre permet sur notre exemple donc de passer de cinq à une seule période d'inactivité comme illustré sur la Figure 5.3, activant un état basse-consommation plus efficace énergétiquement. A noter que sur cette figure, nous avons choisi d'ordonnancer τ' en fin d'intervalle dans l'intervalle 1 et en début d'intervalle dans l'intervalle 4. Nous détaillons ci-dessous comment cette décision est prise lors de l'exécution du système.

$$(PL\ 2) \quad \begin{cases} \text{Minimiser} \sum_k f_k + e_k + fc_k + ec_k \\ \forall k, \sum_{j \in J_k} w_{j,k} \leq m \\ \forall k, \forall j, 0 \leq w_{j,k} \leq 1 \\ \forall j, \sum_{k \in E_j} w_{j,k} \times |I_k| = j.c \\ f_k = \begin{cases} 0 & \text{si } w_k = 1 \\ 1 & \text{sinon} \end{cases} \\ e_k = \begin{cases} 0 & \text{si } w_k = 0 \\ 1 & \text{sinon} \end{cases} \\ fc_k = \begin{cases} 1 & \text{si } f_k = 1 \text{ et } f_{k+1} = 0 \\ 0 & \text{sinon} \end{cases} \\ ec_k = \begin{cases} 1 & \text{si } e_k = 1 \text{ et } e_{k+1} = 0 \\ 0 & \text{sinon} \end{cases} \end{cases}$$

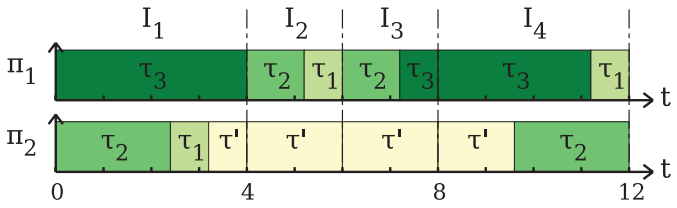

FIGURE 5.3 – Ordonnancement de l'ensemble de tâches 4 avec LPDPM1.

5.1.4 Ordonnancement en-ligne

Une fois connu le poids de chaque tâche, le système doit ordonnancer les tâches à l'intérieur des intervalles. Dans chaque intervalle, l'utilisation globale est égale au nombre de processeurs du système donc tous les intervalles sont ordonnançables en utilisant un algorithme d'ordonnancement multiprocesseur optimal. Comme indiqué au chapitre 4, nous

utilisons FPZL, un algorithme d'ordonnancement multiprocesseur qui est ici optimal car l'exécution du système est divisée en intervalles où tous les travaux possèdent une échéance identique.

Pour réduire la consommation statique, l'algorithme en-ligne doit avoir comme objectif de générer de larges périodes d'inactivité, limitant ainsi le nombre de préemptions de la tâche τ' à l'intérieur des intervalles et entre intervalles. Les poids obtenus par la résolution du programme linéaire ont été optimisés dans cet objectif mais un algorithme en-ligne est néanmoins nécessaire. La solution donnée hors-ligne ne donne en effet pas un ordonnancement complet mais seulement le temps d'exécution de chaque tâche sur chaque intervalle. D'autre part lors de l'exécution du système, les tâches peuvent terminer leur exécution avant leur WCET, ce qui libère du temps processeur qui peut être utilisé pour élargir les périodes d'inactivité existantes et ainsi améliorer les performances énergétiques du système.

Dans cette section, le WCET d'un travail désigne le temps d'exécution du travail, soit $w_{j,k} \times |I_k|$ en utilisant le poids du travail tel que donné par la résolution programme linéaire. Chaque travail peut lors de l'exécution n'utiliser qu'une partie de son WCET. Nous nommons ce temps d'exécution AET (*Actual Execution Time*) et $0 \leq AET \leq WCET$. L'AET d'un travail peut être égal à 0. En effet, comme l'exécution de chaque travail est divisée sur plusieurs intervalles, l'exécution d'un travail peut se terminer sur le premier intervalle où le travail est présent et le temps d'exécution de ce travail sur les intervalles suivant sera nul.

Cette sous-section est divisée en deux parties. Elle traite tout d'abord de la solution théorique lorsque les tâches utilisent leur WCET puis ensuite de la solution pratique lorsque les tâches ne consomment pas la totalité de leur WCET lors de l'exécution.

5.1.4.1 Tâches consommant leur WCET

Dans ce paragraphe, nous faisons l'hypothèse que chaque travail utilise la totalité de son WCET pendant son exécution. C'est l'hypothèse qui est envisagée hors-ligne pour la conception du programme linéaire, nous ferons ensuite l'hypothèse que chaque travail ne consomme pas tout son WCET.

Le temps d'inactivité du système est donc exclusivement représentée par τ'. Ainsi, seul l'ordonnancement de la tâche τ' influence la consommation énergétique du système. L'objectif est donc, en plus de respecter les temps d'exécution de toutes les tâches du système, d'ordonnancer τ' pour augmenter la taille des périodes d'inactivité. Nous avons vu que lors de la conception de la fonction objectif nous avons essayé de maximiser le nombre d'intervalles où le poids de τ' est de 0 ou de 1. Dans ces intervalles, l'ordonnancement de τ' est trivial : soit τ' n'est pas exécutée dans l'intervalle, soit τ' est exécutée durant toute la durée de l'intervalle sur un même processeur.

Le programme linéaire permet de réduire le nombre d'intervalles où la tâche τ' n'est exécutée que durant une partie de l'intervalle (i.e. $0 < w_k < 1$). Mais cette situation est néanmoins susceptible de se produire pour certains intervalles. Dans cette situation, l'ordonnancement de τ' à l'intérieur des intervalles a un impact sur la consommation énergétique.

Ordonnancement de τ' lorsque $0 < w_k < 1$. Pour ces intervalles, l'objectif est d'ordonnancer τ' en début ou fin d'intervalle pour agréger cette exécution de τ' avec une autre exécution d'un intervalle voisin. Calculer s'il est plus efficace d'ordonnancer τ' en début ou en fin d'intervalle peut être fait hors-ligne. Mais ce calcul nécessite de calculer les 2^i possibilités où i est le nombre d'intervalles pour lesquels $0 < w_k < 1$. Cela augmenterait la complexité du programme linéaire hors-ligne pour n'en retirer qu'une faible réduction de la consommation énergétique car l'ordonnancement en-ligne dépend fortement du fait que les travaux utilisent ou non leur WCET lors de l'exécution. Pour cette raison, ce calcul est fait en-ligne.

La décision d'ordonnancer τ' en début ou fin d'intervalle est prise au début de l'intervalle. Si le système possède au moins un processeur dans un état basse-consommation, la tache τ' est alors exécutée en début d'intervalle pour continuer cette période d'inactivité. Dans le cas contraire, l'exécution de τ' est programmée pour la fin de l'intervalle. Nous avons adopté cette solution pour sa simplicité même si elle ne donne pas les meilleurs résultats. En effet, il est possible de décider hors-ligne si τ' doit être ordonnancée en début ou fin d'inactivité mais cela implique une complexité supplémentaire.

Activation des états basse-consommation. Pendant l'exécution du système, ordonnancer τ' signifie activer un état basse-consommation si la taille de la période d'inactivité est suffisante. Pour savoir si un état basse-consommation peut être activé, il est nécessaire de connaître la taille de la période d'inactivité, soit le temps d'exécution de τ'. Cet exécution peut cependant être sur plusieurs intervalles consécutifs. Le système doit alors additionner le poids de τ' de l'intervalle courant avec les poids de τ' des intervalles suivants tant que ce poids est de 1. Lorsque le poids de τ' est strictement inférieur à 1, cela signifie que l'exécution de τ' terminera dans cet intervalle. Et dans ce même intervalle, τ' sera exécutée en début d'intervalle car τ' était en cours d'exécution. Une fois la taille de la période d'inactivité connue, le système a deux choix :

1. Activer l'état basse-consommation le plus économe en énergie dont le BET est inférieur à la taille de la période d'inactivité.
2. Si aucun état basse-consommation ne peut être activé, laisser le processeur inactif durant toute l'exécution de τ'.

Exemple. Nous illustrons cette décision d'ordonnancer τ' en début ou fin d'intervalle à travers un exemple composé de 3 intervalles I_1, I_2 et I_3. Sur cet exemple, nous supposons que le poids de τ' sur ces intervalles est identique et est 0.5. Le BET du processeur est égal à 3 unités de temps. La Figure 5.4 illustre 3 scénarios possibles pour l'ordonnancement de τ' sur l'intervalle suivant les valeurs de w_k dans les intervalles précédent et suivant. Une exécution de τ' avec un fond sombre signifie qu'un état basse-consommation est activé, une exécution de τ' avec un fond clair signifie qu'aucune information n'est disponible sur l'activation ou non d'un état basse-consommation et la dernière couleur de fond représente une exécution de τ' où aucun état basse-consommation n'est activable car la période d'inactivité n'est pas assez large.

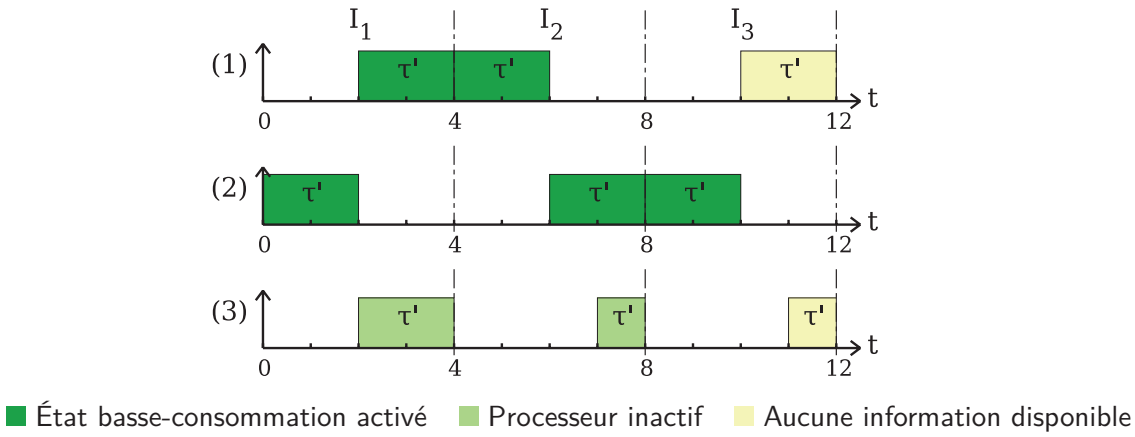

FIGURE 5.4 – Ordonnancement de τ' lorsque $0 < w_k < 1$.

Dans le scénario (1), à $t = 2$, le système peut donc activer un état basse-consommation car la taille de la période d'inactivité partagée sur les intervalles I_1 et I_2 est supérieur au BET. Donc τ' est exécutée au début de l'intervalle I_2. Sur l'intervalle I_3, τ' est ensuite exécutée à la fin de l'intervalle pour qu'elle soit si possible exécutée avec une possible exécution de τ' sur l'intervalle suivant.

Dans les scénarios (2) et (3), nous faisons l'hypothèse que le processeur n'est pas dans un état basse-consommation à $t = 4$. Pour le scénario (3), le processeur est donc simplement inactif à $t = 4$ et τ' est exécutée à la fin de l'intervalle I_2 dans ces 2 scénarios. Nous faisons ensuite l'hypothèse que $w_2 \times |I_2| + w_3 \times |I_3| > BET$ est vrai dans le scénario (2) et faux dans le scénario (3). Dans le premier cas, un état basse-consommation est donc activé quand τ' est exécutée dans I_2 et la période d'inactivité continue au début de l'intervalle I_3. Au contraire, dans le scénario (3), l'exécution de τ' dans I_2 ne permet pas l'activation d'un état basse-consommation. Le processeur reste inactif et l'exécution de τ' est programmée pour la fin de l'intervalle I_3.

5.1.4.2 Tâches se terminant avant d'avoir consommé leur WCET

Cette sous-section traite de la situation où les travaux peuvent terminer leur exécution avant leur WCET, ce qui arrive dans la majorité des cas en pratique. Le temps processeur qui était réservé à ce travail se trouve donc inutilisé et peut être utilisé pour réduire la consommation énergétique. Nous notons ce temps processeur Δt et nous garderons le terme utilisé dans la littérature de *slack time* comme introduit au chapitre 2. Soit r le temps restant dans l'intervalle.

Afin de réduire la consommation énergétique, l'objectif est d'utiliser ce *slack time* pour augmenter la taille des périodes d'inactivité existantes et éviter de créer de courtes périodes d'inactivité. La redistribution du *slack time* dépend de l'état de τ' à l'instant t. Nous notons $\tau'.e$ le temps d'exécution restant de τ' sur l'intervalle courant. Les scénarios suivants sont envisageables lorsqu'une tâche termine son exécution et que du *slack time* est libéré :

— τ' est en cours d'exécution depuis le début de l'intervalle, donc un processeur se trouve dans un état basse-consommation. Le *slack time* peut donc être donné à τ'. Nous avons donc :

$$\tau'.e = \tau'.e + \Delta t \tag{5.10}$$

L'exécution de τ' est donc prolongée et le processeur peut rester dans un état basse-consommation plus longtemps. Par contre, le temps d'exécution de τ' ne peut pas dépasser le temps dans l'intervalle sous peine que sa laxité devienne égale à 0. L'équation 5.10 devient donc :

$$\tau'.e = min(\tau'.e + \Delta t, r) \tag{5.11}$$

Si $r < \tau'.e + \Delta t$, le *slack time* libéré ne peut être donné en totalité à τ' et est donc perdu, nous revenons sur cette possibilité par la suite. A noter que suivant la nouvelle taille de la période d'inactivité, le système peut activer un état basse-consommation plus profond.

— Si τ' n'est pas en cours d'exécution depuis le début de l'intervalle, l'exécution de τ' est prévue pour la fin de l'intervalle. Le comportement de l'algorithme est alors différent suivant si τ' est déjà en cours d'exécution :

1. Si τ' n'est pas en cours d'exécution, le *slack time* peut être donné à τ'. Le temps d'exécution de τ' est donc suivant l'équation (5.11).
2. Si τ' est en cours d'exécution, le *slack time* ne peut être donné à τ' car τ' ne serait alors plus ordonnançable dans l'intervalle, sa laxité deviendrait négative. Le *slack time* est alors perdu.

Perdre le *slack time* signifie qu'il ne peut être donné à τ', donc que l'inactivité du processeur ne sera pas représenté par une exécution de τ'. Cependant, le système peut toujours activer un état basse-consommation au lieu de laisser un processeur inactif. La période d'inactivité sera par contre en dehors d'une exécution de τ' et sa taille n'aura donc pas été optimisée hors-ligne pour une meilleur utilisation des états basse-consommation.

Nous illustrons à l'aide d'un exemple les scénarios présentés ci-dessus concernant l'utilisation du *slack time* lors de l'exécution. Nous avons pour cela ordonnancé l'ensemble de tâches du Tableau 5.2 sur un seul intervalle en Figure 5.5. Nous supposons ici que l'intervalle est de longueur 12. Le tableau donne les WCET et les AET de toutes les tâches de l'ensemble de tâches et le WCET de τ'. Nous faisons l'hypothèse que le processeur π_1 est dans un état basse-consommation au début de l'intervalle à $t = 0$, donc τ' est exécutée au début de l'intervalle.

Lorsque τ_1 termine son exécution à $t = 4$, 2 unités de temps processeur sont libérées. Comme τ' est en cours d'exécution, ce *slack time* est donné à τ' qui a maintenant un temps d'exécution restant de 3. τ' termine ensuite son exécution à $t = 7$ et les autres tâches sont ordonnancées. À $t = 8$, quand τ_2 termine son exécution, une unité de temps processeur est libérée. τ' est donc réactivé avec un temps d'exécution de 1. Comme τ' n'était pas active, son exécution est programmée pour la fin de l'intervalle. Ensuite, lorsque τ_4 termine son

	τ_1	τ_2	τ_3	τ_4	τ'
WCET	6	5	5	3	5
AET	4	4	5	2	

TABLE 5.2 – Ensemble de tâches ordonnancé en Figure 5.5.

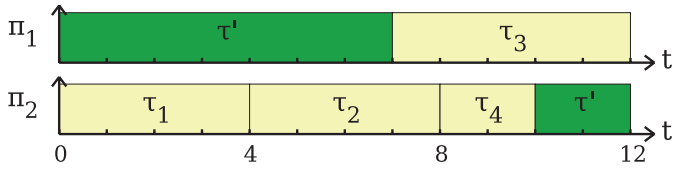

FIGURE 5.5 – Ensemble de tâches du Tableau 5.2 ordonnancé sur un intervalle.

exécution à $t = 10$, le temps d'exécution de τ' gagne une unité de temps et τ' est exécutée. Pour savoir si un état basse-consommation peut être activé lors de cette exécution de τ', le système doit calculer la taille de la période d'inactivité en anticipant l'exécution de τ' sur l'intervalle suivant.

5.2 Minimisation de la consommation statique

Cette section détaille maintenant LPDPM2, un algorithme d'ordonnancement multiprocesseur optimal permettant de minimiser la consommation énergétique. Cet algorithme utilise la même approche que l'algorithme LPDPM1 présenté ci-dessous, mais utilise une fonction objectif différente. En effet, l'hypothèse selon laquelle maximiser la taille moyenne des périodes d'inactivité permet de réduire la consommation statique est une simplification comme l'a illustré le chapitre 3. LPDPM2 résout ce problème en supprimant cette hypothèse et en minimisant directement la consommation énergétique.

Le modélisation du problème dans LPDPM2 est optimale pour optimiser la taille des périodes d'inactivité. En effet, LPDPM2 cherche à minimiser la consommation énergétique et prend donc en considération les caractéristiques des états basse-consommation des processeurs. Pour ce faire, il est nécessaire de calculer la consommation énergétique de chaque période d'inactivité et donc leur longueur afin de pouvoir réduire la consommation énergétique sur l'hyperpériode. L'étape la plus complexe est de calculer la taille de toutes les périodes d'inactivité. Ce point est discuté en premier tandis que le reste de cette section présente la fonction objectif permettant de réduire la consommation statique.

5.2.1 Division de τ' en deux sous-tâches

Comme pour LPDPM1, une période d'inactivité représente une exécution de τ', τ' pouvant être exécutée sur plusieurs intervalles consécutifs. Calculer la taille de toutes les périodes d'inactivité nécessite donc de calculer la durée des exécutions de τ'. Cependant, la modélisation actuelle de τ' permet d'exécuter τ' comme les autres tâches de l'ensemble de

tâches. L'exécution de τ' n'est donc pas restreinte dans l'intervalle et τ' peut également être préemptée. Pour pouvoir calculer la taille des périodes d'inactivité, nous devons restreindre les possibilités d'ordonnancement de τ'.

τ' est une tâche périodique ayant des propriétés identiques à celles des autres tâches du système. τ' peut notamment être ordonnancée comme le système le souhaite à l'intérieur des intervalles. Cependant, sans perdre de gain au niveau énergétique, il est possible de limiter les possibilités d'ordonnancement de τ' à l'intérieur des intervalles. Si le poids de τ' est de 0 ou de 1, aucune marge de manœuvre n'est laissée au système pour ordonnancer τ', nous ne pouvons donc influencer que les intervalles où le poids de τ' est strictement compris entre 0 et 1.

Dans ces intervalles, pour essayer d'agréger cette période d'inactivité avec celles des intervalles voisins, il est nécessaire que la tâche τ' soit exécutée soit en début soit en fin d'intervalle. Il est également nécessaire que le nombre de préemptions de τ' à l'intérieur des intervalles soit limité. Il doit être au maximum de 1, ce qui correspond à la situation où τ' est exécutée au début de l'intervalle, est préemptée et continue son exécution en fin d'intervalle. Toute autre configuration pour ordonnancer τ' ne permettrait pas de diminuer la consommation énergétique.

Le paragraphe précédent nous indique donc que l'ordonnancement de τ' à l'intérieur d'un intervalle peut être représenté, sans aucune perte au niveau énergétique, par deux sous tâches de τ' qui vont être exécutées en début et fin d'intervalle. Nous nommons respectivement τ'_b (*beginning*) et τ'_e (*end*) ces deux sous-tâches. Soit b_k and e_k les poids de τ'_b et τ'_e sur l'intervalle k, b_k et e_k étant deux nombres réels compris entre 0 et 1. Le poids de la tâche τ' dans l'intervalle est donc égal à $b_k + e_k$. Avec ces deux sous-tâches, le programme linéaire du chapitre 4 reste valide.

Cette modélisation de τ' en deux sous-tâches correspond à décision que fait LPDPM1 en-ligne d'ordonnancer τ' en début ou fin d'intervalle. La différence avec LPDPM1 est que LPDPM2 fait ce choix avant l'exécution ce qui permet une meilleure optimisation de la taille des périodes d'inactivité hors-ligne.

La Figure 5.6 illustre la division de τ' en deux sous-tâches. Pour ne pas charger la figure, seules les exécutions de τ' sont représentées sur un seul processeur. Sur cet exemple, l'hyperpériode est divisée en trois intervalles de taille 3. Sur chaque intervalle nous avons représenté le poids de chacune des sous-tâches. Sur cette représentation, nous avons quatre périodes d'inactivité distinctes qui peuvent être regroupées si $b_k + e_k = 1$. Nous voyons maintenant comment obtenir la taille de ces périodes d'inactivité afin de pouvoir calculer leur consommation énergétique.

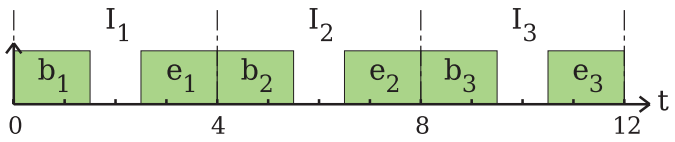

FIGURE 5.6 – Division de τ' en deux sous-tâches.

5.2.2 Calcul de la taille des périodes d'inactivité

Il est nécessaire de calculer la taille de toutes les périodes d'inactivité pour minimiser la consommation statique et activer les états basse-consommation les plus adaptés car calculer la consommation énergétique d'une période nécessite de connaître sa taille. Nous avons vu que sur chaque intervalle, l'exécution de τ' est divisée en deux sous-tâches exécutées en début et en fin d'intervalles. L'exécution de τ' à la fin de l'intervalle $k-1$ et celle de τ' au début de l'intervalle k ne forme en réalité qu'une seule exécution de τ' répartie sur ces deux intervalles $k-1$ et k. Nous avons fait le choix de définir la période d'inactivité de l'intervalle k comme la somme de l'exécution de τ' en fin d'intervalle k et de l'exécution de τ' au début de l'intervalle $k+1$. La période d'inactivité de l'intervalle k peut par conséquent commencer à l'intervalle $k+1$ si $e_k = 0$.

Soit p_k la taille de la période d'inactivité de l'intervalle k. Cette période d'inactivité inclut les deux exécutions de τ' mentionnées ci-dessus, soit les e_k et b_{k+1}. Quand le poids de τ' est de 1 sur l'intervalle $k+1$, les deux périodes d'inactivité des intervalles k et $k+1$ ne forment qu'une seule période d'inactivité. Formellement, nous définissons p_k comme :

$$p_k = \begin{cases} e_k \times |I_k| + b_{k+1} \times |I_{k+1}| + p_{k+1} & \text{si } b_{k+1} + e_{k+1} = 1 \\ e_k \times |I_k| + b_{k+1} \times |I_{k+1}| & \text{sinon} \end{cases} \tag{5.12}$$

Donc la période d'inactivité de l'intervalle inclut les deux sous-tâches caractérisées par les poids e_k et b_{k+1} mais aussi la période d'inactivité de l'intervalle $k+1$ si le poids de l'intervalle $k+1$ est de 1. Dans ce dernier cas, la période d'inactivité de l'intervalle $k+1$ n'a plus lieu d'être. Nous introduisons alors une nouvelle variable réelle positive q_k pour représenter la taille des périodes d'inactivité. q_k est telle que :

$$q_k = \begin{cases} p_k & \text{si } b_k + e_k \neq 1 \\ 0 & \text{sinon} \end{cases} \tag{5.13}$$

Ainsi, la période d'inactivité de l'intervalle k est uniquement conservé si le poids de τ' sur cet intervalle est différent de 1. Dans le cas contraire, cette période d'inactivité est déjà incluse dans celle de l'intervalle $k-1$ et sa taille est donc de 0. Cette modélisation permet de calculer la taille de toutes les périodes d'inactivité, mêmes celles comprises sur plusieurs intervalles successifs. La somme de toutes les exécutions de τ' correspond maintenant à la somme des tailles des périodes d'inactivité et l'équation suivante est vérifiée :

$$\sum_k q_k = w_k * |I_k| \tag{5.14}$$

5.2.2.1 Premier et dernier intervalles

La modélisation précédente ne tient pas compte des spécificités du premier et du dernier intervalle de l'hyperpériode. Comme le premier intervalle de l'hyperpériode ne possède pas d'intervalle précédent, nous devons donc introduire une nouvelle variable pour représenter la taille de la première période d'inactivité de l'hyperpériode. Cette variable p_0 est telle que :

$$p_0 = b_0 * |I_0| \tag{5.15}$$

De même pour le dernier intervalle $k-1$, la taille de sa période d'inactivité est de :

$$p_{k-1} = e_{k-1} * |I_{k-1}| \tag{5.16}$$

Ces deux périodes d'inactivité peuvent se regrouper avec les périodes d'inactivité de leur intervalle voisin si respectivement $b_0 + e_0 = 1$ et $b_{k-1} + e_{k-1} = 1$. Il serait également possible de modéliser l'intervalle I_0 comme l'intervalle suivant de l'intervalle I_{k-1} mais nous avons explicitement choisi de travailler sur une seule hyperpériode.

5.2.2.2 Exemple

Pour illustrer les notations, la Figure 5.7 représente l'ordonnancement de τ' sur un exemple où $H = 18$. Le temps d'exécution de τ' sur cette hyperpériode est de 9. Nous faisons l'hypothèse que l'hyperpériode comprend 6 intervalles de taille identique $|I_k| = 3$. Les valeurs de toutes les variables sont indiquées dans le Tableau 5.3. Sur cet exemple, 4 périodes d'inactivité de taille 5, 2, 1 et sont créées aux intervalles 1, 3, 4 et 5.

La période d'inactivité de l'intervalle 1 commence avec l'exécution de e_1 et se poursuit sur l'intervalle 2 jusqu'à l'exécution de b_3 car $b_2 + e_2 = 1$. Pour les autres périodes d'inactivité, elles ne sont que sur un seul intervalle. A noter que la période d'inactivité de l'intervalle 5 se situe uniquement sur l'intervalle 6 car $e_5 = 0$ et $b_6 > 0$.

k	0	1	2	3	4	5	6
$b_k * \|I_k\|$	0	0	1.5	1	1	0	1
$e_k * \|I_k\|$	0	1	1.5	1	1	0	0
p_k	0	5	2.5	2	1	1	0
q_k	0	5	0	2	1	1	0

TABLE 5.3 – Exemple avec 4 intervalles, $\tau'.c = 8$.

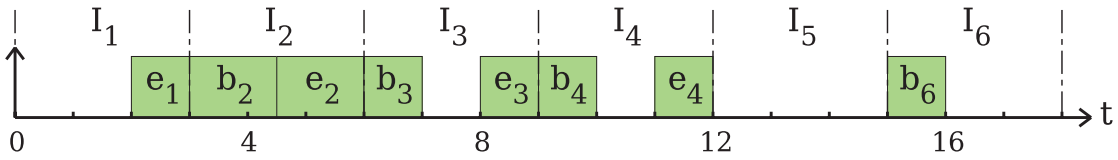

FIGURE 5.7 – Ordonnancement de τ' à partir du tableau 5.3.

5.2.3 Fonction d'optimisation

Pour chaque période d'inactivité dont la taille (i.e. q_k) est strictement supérieure à 0, un état basse-consommation peut être activé. L'état basse-consommation choisi est celui qui entraine la consommation énergétique minimale sur la période d'inactivité. Afin d'intégrer

dans le programme linéaire le choix de l'état basse-consommation, nous définissons $LP_{s,k}$ une variable binaire. $LP_{s,k} = 1$ si l'état basse-consommation s est utilisé par la période d'inactivité de l'intervalle k et est égale à 0 sinon :

$$LP_{s,k} = \begin{cases} 1 & \text{si l'état basse-consommation } s \text{ est utilisé} \\ & \text{par la période d'inactivité de l'intervalle } k \\ 0 & \text{sinon} \end{cases} \tag{5.17}$$

Chaque période d'inactivité de longueur strictement supérieure à 0 doit activer un et un seul état basse-consommation, nous avons par conséquent :

$$\forall k,\ \sum_s LP_{s,k} = \begin{cases} 0 & \text{si } q_k = 0 \\ 1 & \text{sinon} \end{cases} \tag{5.18}$$

Une période d'inactivité peut par contre n'être pas suffisament large pour activer un état basse-consommation. Il est donc nécessaire d'ajouter dans la liste des états basse-consommation l'état actif du processeur. Cet état basse-consommation a un BET nul et peut par conséquent toujours être activé.

Un état basse-consommation ne peut être activé que si la taille de la période d'inactivité est plus grande que son BET. Une contrainte supplémentaire est donc nécessaire :

$$\forall k, \forall s,\ LP_{s,k} = 0 \text{ si } q_k \leq BET_s \tag{5.19}$$

Comme présenté au chapitre 2, la consommation énergétique dans l'état basse-consommation s est $Cons_s$ tandis que la pénalité pour revenir de cet état basse-consommation est Pen_s. La consommation énergétique de la période d'inactivité de l'intervalle k est donc :

$$P_k = \sum_s LP_{s,k}(Cons_s \times q_k + Pen_s) \tag{5.20}$$

Toutes les notations introduites dans cette section sont résumées dans le Tableau 5.4.

b_k	Poids de la sous-tâche τ'_b sur l'intervalle k
e_k	Poids de la sous-tâche τ'_e sur l'intervalle k
p_k	Taille de la période d'inactivité comprenant e_k et b_{k+1}
q_k	Taille de la période d'inactivité de l'intervalle k
$LP_{s,k}$	1 si la période d'inactivité de l'intervalle k active l'état basse-consommation s, 0 sinon
P_k	Consommation énergétique de la période d'inactivité de l'intervalle k

TABLE 5.4 – Récapitulation des notations pour LPDPM2.

Afin de réduire la consommation énergétique de toutes les périodes d'inactivité, la fonction objectif finale de notre programme linéaire en variables mixtes est :

$$\text{Minimiser } \sum_k P_k \tag{5.21}$$

Cette fonction objectif permet de minimiser la consommation énergétique lorsque la tâche τ' est exécutée et ne prend pas en compte la consommation énergétique lorsque les autres tâches sont en cours d'exécution. En effet, la consommation énergétique de ces tâches est connue car leur temps d'exécution est fixe et la consommation énergétique lors de l'exécution est constante. Donc la consommation énergétique induite par ces tâches est une constante qui ne peut être réduite par l'algorithme d'ordonnancement. Le programme linéaire (PL 3) est maintenant complet.

$$(PL\ 3) \qquad \begin{cases} \text{Minimiser } \sum_k P_k \\ \forall k, \ \sum_{j \in J_k} w_{j,k} \leq m \\ \forall k, \forall j, \ 0 \leq w_{j,k} \leq 1 \\ \forall j, \ \sum_{k \in E_j} w_{j,k} \times |I_k| = j.c \\ P_k = \sum_s LP_{s,k}(Cons_s \times q_k + Pen_s) \end{cases}$$

Nous avons ordonnancé sur deux processeurs l'ensemble de tâches 4 du Tableau 5.5 en Figures 5.8 et 5.9 avec respectivement LPDPM1 et LPDPM2. Pour LPDPM1, nous avons utilisé l'algorithme d'ordonnancement en-ligne proposé au paragraphe 5.1.4 pour ordonnancer τ' lorsque $0 < w_k < 1$. Pour LPDPM2, les intervalles sont ordonnancés en utilisant FPZL suivant les poids obtenus en résolvant le programme linéaire (PL 3). Sur cet exemple, nous observons que LPDPM1 génère 3 périodes d'inactivité alors que LPDPM2 n'en produit que 2 ce qui permet d'activer des états basse-consommation plus efficaces.

	τ_1	τ_2	τ_3	τ_4	τ_5
WCET	3.5	1	1	1	1
Période	4	6	6	8	8

TABLE 5.5 – Ensemble de tâches 5.

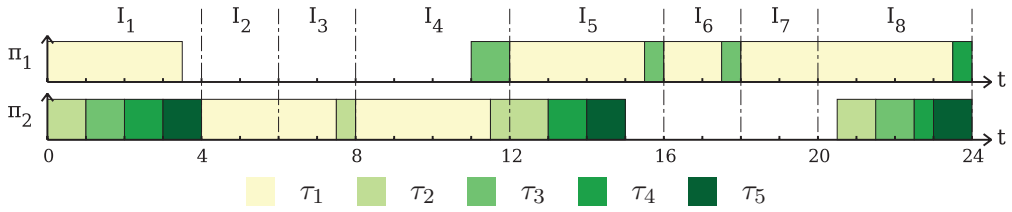

FIGURE 5.8 – Ordonnancement de l'ensemble de tâches 5 avec LPDPM2.

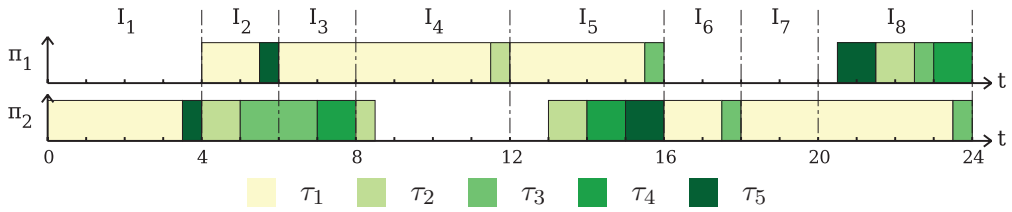

FIGURE 5.9 – Ordonnancement de l'ensemble de tâches 5 avec LPDPM1.

5.2.4 Ordonnancement en-ligne

Nous détaillons dans cette section l'algorithme en-ligne ordonnançant les tâches à l'intérieur des intervalles. Comme pour LPDPM1, cette partie est divisée en deux, la première faisant l'hypothèse que chaque travail utilise tout son WCET tandis que dans la seconde les travaux peuvent terminer leur exécution plus tôt que prévu.

5.2.4.1 Tâches consommant leur WCET

Comme τ' est déjà divisé en deux sous-tâches, l'ordonnancement des tâches dans les intervalles suit l'algorithme d'ordonnancement multiprocesseur FPZL sans qu'il y ait besoin d'intervenir pour étendre les périodes d'inactivité. Les deux sous-tâches τ'_e et τ'_b se voient affecter respectivement les priorités les plus faibles et plus élevées pour être exécutées respectivement en fin et début d'intervalle.

Lors de l'exécution, ces deux tâches ne peuvent pas être préemptées. En effet, τ'_b a la priorité la plus importante et le seul scénario entrainant une préemption de τ'_b surviendrait lorsque la laxité d'une tâche devient nulle et qu'aucun autre processeur ne peut être préempté. Cette situation est impossible car il faudrait que tous les autres processeurs exécutent des tâches ayant une laxité de 0 alors que τ' a encore du temps d'exécution restant. Pour τ'_e, ayant la plus faible priorité, elle ne peut être exécutée que lorsque sa laxité atteint 0 et ne peut donc pas être préemptée par la suite.

5.2.4.2 Tâches se terminant avant d'avoir consommé leur WCET

Lors de l'exécution du système, les travaux terminent leur exécution avec leur WCET et libère du *slack time*. Comme pour la section précédente, l'objectif est d'utiliser le *slack time* Δt libéré pour réduire la consommation énergétique en allongeant les périodes d'inactivité existantes. Concrètement, il s'agit augmenter les temps d'exécution des deux sous-tâches de τ'. Cela dépend de l'état de τ' à l'instant t lorsque le *slack time* est libéré :

1. Si τ'_b est active à l'instant t, alors le *slack time* est donné à τ'_b. Ce n'est possible que tant que la laxité de τ'_b et τ'_e combinée reste positive.
2. Si τ'_b n'est pas active, le *slack time* est donné à τ'_e tant que sa laxité reste positive pour être utilisée en fin d'intervalle.
3. Sinon, le *slack time* n'est pas utilisable par τ'_b ou τ'_e.

Comme pour LPDPM1, il existe des situations où le *slack time* ne peut être utilisé pour agrandir une période d'inactivité existante car la laxité combinée de τ'_b et τ'_e ne peut être négative. D'autres états basse-consommation peuvent néanmoins être activés sur d'autres processeurs mais la taille de ces périodes d'inactivité n'aura pas été optimisé hors-ligne.

5.3 Évaluation

Dans cette section, nous évaluons les différences entre les performances de LPDPM1 et LPDPM2 ainsi que les différences entre LPDPM1, LPDPM2 et certains algorithmes d'ordonnancement multiprocesseurs temps réel optimaux. À l'aide d'un simulateur développé dans le cadre de cette thèse, nous comparons l'utilisation des états basse-consommation, la taille des périodes d'inactivité, la consommation énergétique et le nombre de préemptions de chaque algorithme d'ordonnancement.

5.3.1 Environnement de simulation

Nous avons développé dans le cadre de cette thèse un outil permettant de simuler l'ordonnancement d'ensembles de tâches. Cet outil prend en entrée un nombre de processeurs, des caractéristiques des processeurs, un ensemble de tâches et un algorithme d'ordonnancement. Il donne en résultat le graphe d'ordonnancement sur la période de temps voulu et différentes données sur l'ordonnancement généré comme par exemple la consommation énergétique mais aussi le nombre de préemptions, l'optimalité de la solution du programme linéaire ou les états basse-consommation utilisés. Pour ne pas surcharger cette section, nous n'avons sélectionné que les résultats qui nous apparaissent les plus importants.

Ensembles de tâches. Pour les simulations effectuées dans le cadre de ce chapitre, nous avons généré de manière aléatoire des ensembles de tâches que nous avons ensuite ordonnancés sur des systèmes à 2, 4 ou 8 processeurs. Chaque ensemble de tâches comprend respectivement 5, 10 et 20 tâches suivant le nombre de processeurs utilisés.

Chaque ensemble de tâches est ordonnancé sur 2 hyperpériodes. Nous avons fait ce choix pour intégrer dans notre calcul de la consommation énergétique une période d'inactivité qui pourrait intervenir entre deux hyperpériodes consécutives, et ce même si nous avons choisi de ne pas intégrer ce problème dans notre solution. Dans le cadre de ces simulations, nous avons fait le choix d'ordonnancer les tâches en utilisant durant toute la simulation le WCET des tâches, comme le font la majorité des travaux (e.g. [BFBA09]). Nous reviendrons sur ce point au chapitre 6 pour des systèmes temps réel à criticité mixte où nous utiliserons l'AET des tâches de plus faible criticité.

Pour chaque ensemble de tâches, l'utilisation individuelle de chaque tâche est calculée suivant une distribution uniforme en utilisant l'algorithme UUniFast-Discard de Davis et Burns [DB11b] qui est une version pour systèmes multiprocesseurs de l'algorithme UUniFast de Bini et Buttazzo [BB04].

Les périodes de toutes les tâches ont été générées de manière aléatoire entre $10ms$ et $100ms$ suivant une distribution uniforme, les valeurs des périodes devant être des multiples de

$10ms$. Les ensembles de tâches avec des périodes plus larges que $10s$ ont été rejetés lors de la génération pour rester dans les cadres de grandeurs courants des systèmes industriels. Pour chaque ensemble de tâches, l'utilisation globale est fixée entre $m-1$ et m. 500 ensembles de tâches sont générés pour chaque utilisation globale différente.

Algorithme d'ordonnancement. Nous comparons les deux algorithmes présentés dans ce chapitre LPDPM1 et LPDPM2 à deux algorithmes d'ordonnancement temps réel multi-processeurs optimaux RUN [RLM+11] et U-EDF [NBGM11]. Ces deux solutions ont pour objectif de réduire le nombre de préemptions et de migrations. Nous les avons choisis car il n'existe pas d'algorithme d'ordonnancement multiprocesseur optimal ayant pour objectif de réduire la consommation statique. AsDPM [BFBA09], le seul algorithme d'ordonnancement multiprocesseur global dont l'objectif est de réduire la consommation statique n'est en effet pas optimal. Ces deux algorithmes d'ordonnancement sont également deux algorithmes de référence concernant le nombre de préemptions et de migrations. Notre implémentation de l'algorithme d'ordonnancement *RUN* ajoute une tâche représentant le temps d'inactivité, de façon similaire à LPDPM avec τ', pour que l'utilisation globale de l'ensemble de tâches soit égale au nombre de processeur.

Processeur. Dans le cadre de cette simulation, nous utilisons un processeur avec quatre états basse-consommation dont les consommations énergétiques et les délais de transition sont détaillés dans le Tableau 5.6. Nous utilisons une consommation énergétique normalisée et nous supposons que la consommation énergétique dans l'état actif est de 1. Ces valeurs sont issues de l'analyse que nous avons faite d'une gamme de processeurs présentés au chapitre 2. Nous faisons ensuite l'hypothèse que la consommation énergétique lors de la transition d'un état basse-consommation vers l'état actif est linéaire comme Awan et Petters [AP11]. Le BET de chaque état basse-consommation est par conséquent égal à son délai de transition. Un état basse-consommation peut être activé dès que la taille de la période d'inactivité est supérieure au délai de transition de cet état basse-consommation.

Mode	Consommation énergétique	Délai de transition
Run	1	
Sleep	0.5	$0.01ms$
Stop	0.1	$2ms$
Standby	0.00001	$10ms$

Table 5.6 – États basse-consommation utilisés pour l'évaluation.

La valeur du délai de transition pour l'état basse-consommation le plus profond est de $10ms$, et est donc égale à la valeur minimale des périodes des tâches. Utiliser cet état basse-consommation nécessite par conséquent un algorithme d'ordonnancement spécialement conçu pour optimiser la taille des périodes d'inactivité.

5.3.2 Résolution du programme linéaire

Nombre de processeurs	LPDPM1	LPDPM2
2	100	23
4	99	0.1
8	88	0

Table 5.7 – Pourcentage de solutions optimales.

Chaque programme linéaire est résolu en utilisant le solveur IBM ILOG CPLEX 12.4 [CPL]. Pour chaque programme linéaire, 1 minute est attribuée à CPLEX pour résoudre le problème. Ces tests ont été effectués sur une machine avec 48 cœurs et 64Go de RAM et 5 programmes linéaires sont résolus en parallèle. Nous avons utilisé la version 12.4 de CPLEX. Le Tableau 5.7 donne le pourcentage de résultats optimaux pour LPDPM1 et LPDPM2 suivant le nombre de processeurs.

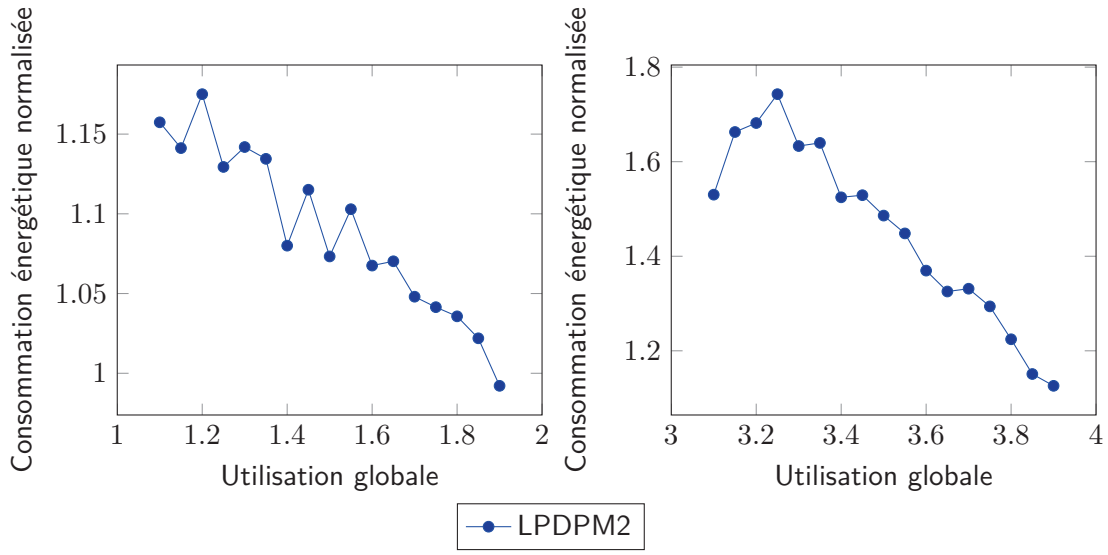

Figure 5.10 – Consommation énergétique normalisée de LPDPM2 par rapport à LPDPM1 avec 2 (a) et 4 (b) processeurs.

5.3.3 Comparaison entre LPDPM1 et LPDPM2

L'objectif de cette sous-section est d'évaluer les différences entre les deux algorithmes d'ordonnancement LPDPM1 et LPDPM2.

La Figure 5.10 présente la consommation énergétique de LPDPM2 par rapport à celle de LPDPM1 avec respectivement deux et quatre processeurs. Nous avons seulement calculé la consommation énergétique lorsque les processeurs sont inactifs ou dans un état basse-consommation (i.e. lorsque la tâche τ' est en cours d'exécution). En effet, la consommation

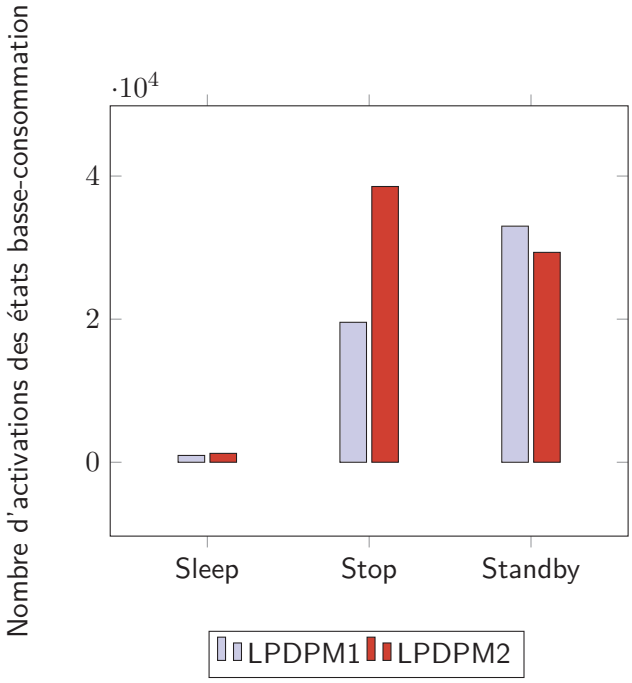

Figure 5.11 – Utilisation des états basse-consommation (2 processeurs).

énergétique lors de l'exécution des autres tâches est identique pour LPDPM1 et LPDPM2, intégrer leur consommation énergétique ne serait donc pas utile pour illustrer les différences entre les algorithmes d'ordonnancement. Que ce soit avec deux ou quatre processeurs, la consommation énergétique de LPDPM2 est toujours supérieure à celle de LPDPM1 lorsque l'utilisation globale est faible. L'écart entre les deux algorithmes d'ordonnancement se réduit lorsque l'utilisation globale augmente. LPDPM2 obtient même une consommation énergétique plus faible que LPDPM1 pour une utilisation globale de 1.9. Le fait que l'écart entre les deux algorithmes d'ordonnancement se réduise lorsque l'utilisation globale augmente peut s'expliquer par le fait qu'une utilisation globale faible laisse plus de temps processeur libre pour activer des états basse-consommation. Optimiser la taille des périodes d'inactivité est donc plus important lorsque l'utilisation globale est faible car les possibilités offertes par l'utilisation des états basse-consommation sont les plus importantes.

Pour avoir plus d'informations sur la différence entre LPDPM1 et LPDPM2, la Figure 5.11 présente l'utilisation des états basse-consommation par ces deux algorithmes d'ordonnancement. Cette figure montre que LPDPM1 fait un meilleur usage des état basse-consommation en utilisant plus l'état *Standby* que le mode *Stop* au contraire de LPDPM2.

Les résultats montrent donc que les performances de LPDPM2 sont au mieux égales à celles de LPDPM1 alors que LPDPM2 est théoriquement plus efficace que LPDPM1 car il calcule la consommation énergétique de chaque période d'inactivité. Nous pouvons cependant proposer plusieurs explications pour justifier les meilleurs résultats obtenus par LPDPM1.

Les performances de LPDPM1 peuvent tout d'abord s'expliquer par le fait que LPDPM1

obtient un poids strictement compris entre 0 et 1 pour la tâche τ' pour seulement 13% des intervalles. Ces intervalles sont optimisés en ligne par LPDPM1 pour que τ' soit ordonnancée en début ou fin d'intervalle. Comme ces intervalles sont censés être favorables à LPDPM2, leur nombre limité peut expliquer les meilleures performances de LPDPM1 par rapport à de LPDPM2.

Ces résultats peuvent également s'expliquer par le fait que les résultats donnés par LPDPM2 ne sont pas optimaux comme illustré au Tableau 5.7. Comme le temps de résolution laissé à CPLEX est relativement faible, la solution optimale du programme linéaire n'est obtenue que pour LPDPM1. Avec LPDPM2, le nombre de solutions optimales est presque nul quand le nombre de processeurs est supérieur ou égal à 4. La différence entre LPDPM1 et LPDPM2 était attendue, LPDPM1 étant plus simple que LPDPM2, mais l'écart entre les deux solutions est néanmoins important. La différence entre les programmes linéaires de LPDPM1 et LPDPM2 est la division pour LPDPM2 de τ' en deux sous-tâches τ'_b et τ'_e pour savoir si τ' doit être ordonnancée en début ou fin d'intervalle. Nous avons vu que le programme linéaire de LPDPM1 est par conséquent seulement une heuristique vis-à-vis de la réduction de la consommation énergétique. Cette division de τ' augmente la complexité du programme linéaire de LPDPM2 mais permet d'avoir une modélisation correcte de la longueur des périodes d'inactivité. En effet, LPDPM2 réalise un calcul exact de consommation énergétique, ce qui nécessite de connaître la longueur des périodes d'inactivité et donc de faire le choix d'ordonnancer τ'_b et τ'_e en début ou fin d'intervalle dans le programme linéaire. Nous avons également effectué des simulations en allouant davantage de temps au solveur pour trouver une solution, en augmentant le temps de résolution à plusieurs heures, mais le nombre de solutions optimales pour LPDPM2 reste proche de 0 avec 4 et 8 processeurs. En conclusion, ces résultats tendent à montrer que la simplification faite par LPDPM1 est appropriée.

5.3.4 Comparaison avec les algorithmes existants

Dans cette section, nous comparons LPDPM1 à RUN et U-EDF. Nous n'avons conservé que LPDPM1 car nous avons vu que LPDPM1 a de meilleures performances énergétique que LPDPM2. Nous évaluons trois résultats pour chaque environnement de simulation et pour chaque algorithme d'ordonnancement : le nombre de périodes d'inactivité, la consommation énergétique et le nombre de préemptions.

Nombre de processeurs	LPDPM1	U-EDF	RUN
2	53×10^3	170×10^3	259×10^3
4	11×10^3	48×10^3	132×10^3
8	7×10^3	50×10^3	236×10^3

Table 5.8 – Nombre moyen de périodes d'inactivité.

5.3.4.1 Nombre de périodes d'inactivité

Le Tableau 5.8 donne la taille moyenne de toutes les périodes d'inactivité générées par un algorithme d'ordonnancement sur les trois environnements de simulation utilisés. LPDPM1 génère toujours moins de périodes d'inactivité que les autres algorithmes d'ordonnancement et ce quelque soit le nombre de processeurs. Entre les deux autres algorithmes d'ordonnancement, U-EDF génère toujours moins de périodes d'inactivité que RUN.

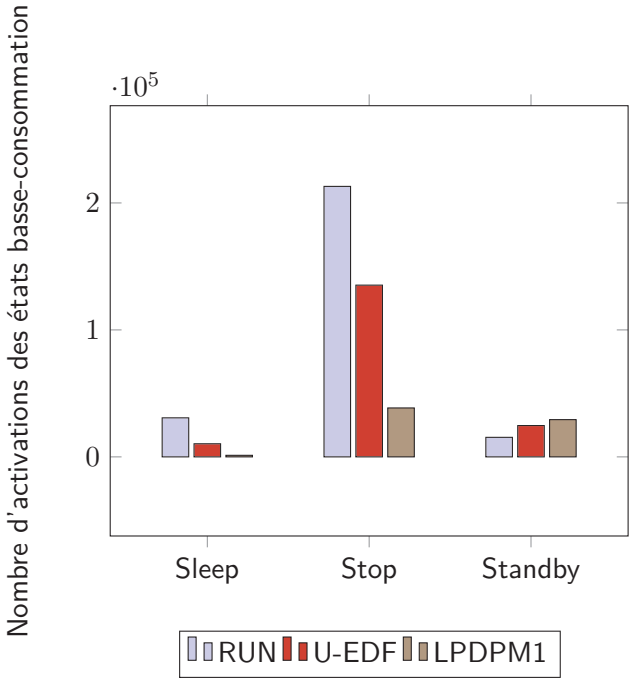

Figure 5.12 – Utilisation des états basse-consommation (2 processeurs).

5.3.4.2 Utilisation des états basse-consommation

Les Figures 5.12 et 5.13 représentent l'utilisation des états basse-consommation pour chaque algorithme d'ordonnancement et pour respectivement les environnements de simulation avec 2 et 4 processeurs. En ordonnée de ces figures est donné le nombre de fois où l'état basse-consommation est utilisé pour chaque processeur. Chaque activation d'un état basse-consommation correspondant à une période d'inactivité. Dans ces figures ne sont pas séparées les différentes utilisations globales, tous les ordonnancements générés pour chaque algorithme d'ordonnancement sont utilisés.

Ces figures montrent que le nombre de périodes d'inactivité pour RUN et U-EDF est bien plus important que pour notre solution, ce qui conforte les résultats du Tableau 5.8. RUN et U-EDF utilisent majoritairement l'état *Stop* tandis que LPDPM1 n'utilise que très rarement l'état basse-consommation le moins économe en énergie *Sleep* pour utiliser en priorité les états *Stop* et *Standby*.

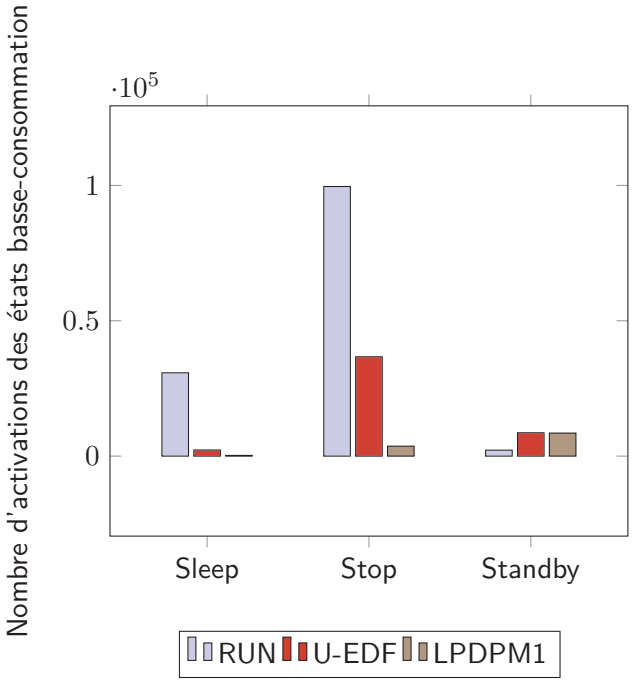

Figure 5.13 – Utilisation des états basse-consommation (4 processeurs).

5.3.4.3 Consommation énergétique

Les Figures 5.14 et 5.15 représentent les consommations énergétiques des trois environnements de simulations, à savoir avec respectivement 2, 4 et 8 processeurs. Pour chaque figure, l'axe des abscisses représente l'utilisation globale du système entre $m-1$ et m. Chaque figure donne la consommation énergétique normalisée où la consommation énergétique de LPDPM1 est toujours égale à 1.

Lorsque l'utilisation globale est faible, LPDPM1 est jusqu'à 10 fois plus efficace que les autres solutions. Cette différence se réduit lorsque l'utilisation globale augmente. En effet, le temps d'inactivité utilisable par les états basse-consommation est alors réduit et LPDPM1 ne peut plus créer de larges périodes d'inactivité. La différence importante entre LPDPM1 et les deux autres algorithmes lorsque l'utilisation est faible peut s'expliquer par le fait que le temps d'inactivité disponible pour créer de larges périodes d'inactivité est important, un processeur peut presque être toujours dans un état basse-consommation. Nos deux solutions créent alors de larges périodes d'inactivité permettant d'utiliser l'état basse-consommation le plus important où la consommation énergétique est proche de 0. Au contraire, RUN et U-EDF créent toujours de courtes périodes d'inactivité.

5.3.4.4 Préemptions

La Figure 5.16 donne le nombre moyen de préemptions pour chaque ordonnanceur suivant l'utilisation globale pour 2 et 4 processeurs. Nous avons compté le nombre de préemptions

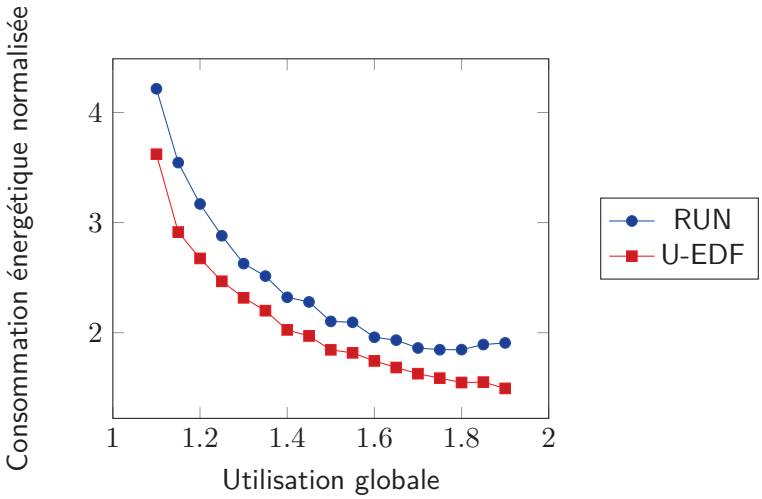

Figure 5.14 – Consommation énergétique normalisée de RUN et U-EDF par rapport à LPDPM1 (2 processeurs).

engendrées sur l'ensemble des tâches, nous ajoutons une préemption quand une tâche active à l'instant t sur un processeur c n'est plus active à l'instant $t + \epsilon$ sur ce même processeur. Donc une préemption représente ici à la fois la terminaison d'une exécution d'une tâche mais aussi une préemption comme définie au chapitre 2. Les migrations sont ici ignorées.

Le nombre de préemptions générées par LPDPM1 est similaire à ceux de U-EDF, et toujours mieux de 1.4 fois ceux de RUN. Ce qui nous permet de conclure que les préemptions ont une influence sur l'efficacité énergétique de LPDPM1 comparable à celle sur RUN et U-EDF. En effet, même si la pénalité énergétique engendrée par une préemption est plus importante que prévue, ces pénalités seront également subies par RUN et U-EDF qui ne verront pas leur consommation énergétique s'améliorer par rapport à LPDPM.

Notre environnement de simulation nous a donc permis d'observer que LPDPM1 est plus efficace énergétiquement que les algorithmes d'ordonnancement multiprocesseurs existants, et ce quel que soit les hypothèses de travail utilisées. Nous avons également effectués ces tests en prenant et LPDPM1 et LPDPM2 et nous observons des résultats similaires. Néanmoins, comme souligné en section 5.3.3, les performances de LPDPM1 restent supérieures à celles de LPDPM2 concernant la consommation énergétique et similaires concernant le nombre de préemptions.

Nous avons également effectué ces simulations avec 16 processeurs. Les différences entre les algorithmes d'ordonnancement restent similaires et nos solutions sont toujours plus efficaces énergétiquement.

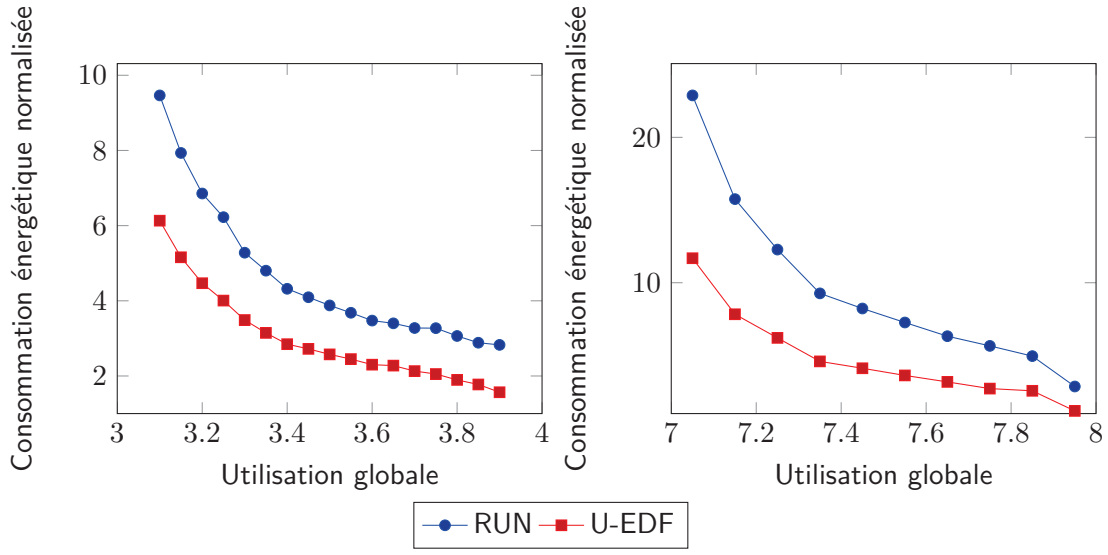

Figure 5.15 – Consommation énergétique normalisée de RUN et U-EDF par rapport à LPDPM1 avec 4 (a) et 8 (b) processeurs.

5.4 Conclusion

Ce chapitre a présenté deux algorithmes d'ordonnancement multiprocesseur temps réel optimaux, LPDPM1 et LPDPM2, permettant de réduire la consommation statique. Ces deux algorithmes sont basés sur l'approche décrite au chapitre 4. Ils génèrent un ordonnancement hors-ligne permettant de réduire la consommation énergétique en utilisant les états basse-consommation des processeurs. En-ligne, lorsque les tâches finissent leur exécution plus tôt que prévue, plusieurs stratégies ont été présentées afin d'étendre les périodes d'inactivité existantes.

La différence entre ces deux algorithmes se situe dans la fonction objectif permettant au programme linéaire utilisé d'optimiser la consommation énergétique. LPDPM1 a pour objectif de minimiser le nombre de périodes d'inactivité tandis LPDPM2 cherche à minimiser la consommation statique. L'avantage de LPDPM1 est que le résultat calculé hors-ligne peut être utilisé sur n'importe quel processeur. Au contraire, le résultat de LPDPM2 est optimisé pour un processeur en particulier mais est théoriquement plus performant au prix d'une plus importante complexité.

Les évaluations montrent que LPDPM2 est au mieux aussi performant que LPDPM1 car la solution du programme linéaire pour LPDPM2 n'est pas optimale. Ces évaluations montrent néanmoins que ces deux solutions sont largement plus efficaces que les algorithmes d'ordonnancement optimaux existants RUN et U-EDF. La consommation énergétique lorsque les processeurs n'exécutent pas les tâches est en-effet réduite d'un facteur 10 lorsque le temps d'inactivité dans le système est maximal. Les simulations montrent également que LPDPM1 et LPDPM2 n'augmentent pas le nombre de préemptions en contrepartie de leur meilleur efficacité énergétique. Pour valider leur efficacité énergétique de façon expérimentale, une

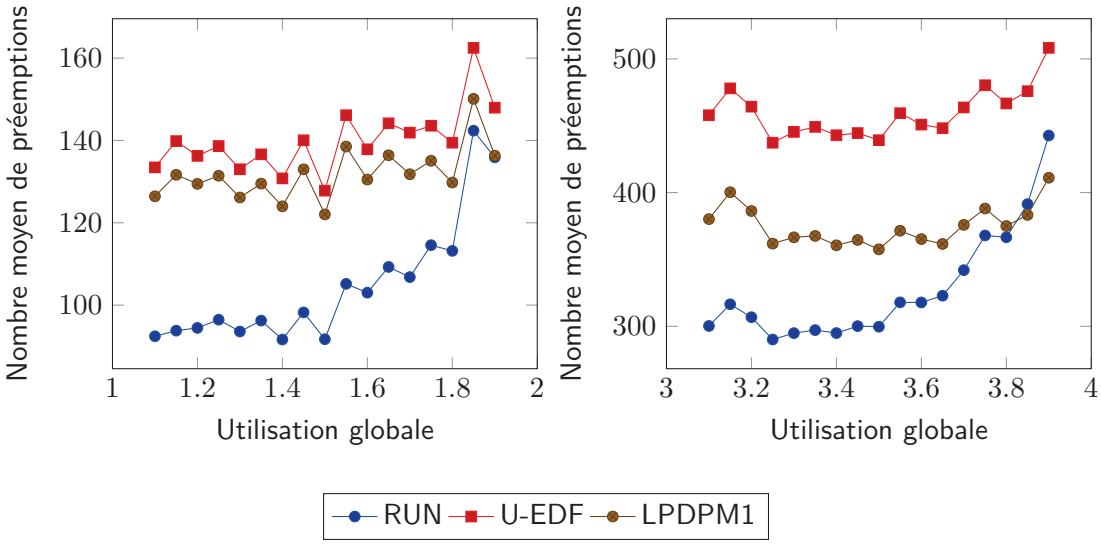

Figure 5.16 – Nombre moyen de préemptions de RUN, U-EDF et LPDPM1 avec 2 (a) et 4 (b) processeurs.

perspective serait d'implémenter LPDPM1 et LPDPM2 dans un noyau existant.

Le chapitre suivant présente LPDPM-MC, un algorithme d'ordonnancement multiprocesseur optimal pour réduire la consommation énergétique des systèmes temps réel à criticité mixte exploitant le fait que les tâches à faible criticité peuvent tolérer des dépassements d'échéances.

CHAPITRE 6

Systèmes temps réel à criticité mixte

Sommaire

Nous traitons dans ce chapitre des systèmes temps réel à criticité mixte. Comme vu au chapitre 4, l'approche suivie est basée sur celle utilisée pour les systèmes temps réel dur mais avec des spécificités relatives à la multicriticalité. Ce chapitre détaille ces spécificités respectivement dans les parties hors-ligne et en-ligne et termine par une évaluation de cet algorithme d'ordonnancement comparé à la solution présentée au chapitre 5 pour des systèmes temps réel dur. L'algorithme décrit dans cette section est nommé LPDPM-MC pour *Linear Programing Dynamic Power Management for Mixed Criticality*.

6.1 Compromis entre dépassements d'échéances et gain énergétique

Cette section présente LPDPM-MC, un algorithme d'ordonnancement optimal dont l'objectif est de réduire la consommation statique des systèmes temps réel à criticité mixte en effectuant un compromis entre la réduction de la consommation statique et le respect des échéances des tâches à faible criticité.

6.1.1 Introduction

Les systèmes temps réel à criticité mixte ont été introduits par Vestal [Ves07] et Baruah et al. ([BBD+10, BBD11]). Par rapport à l'ensemble de tâches de Liu et Layland [LL73], chaque tâche d'un systèmes temps réel à criticité mixte peut avoir plusieurs niveaux de criticité. Comme vu au chapitre 2, nous supposons deux niveaux de criticité (*HIGH* et *LOW*) comme la majorité des travaux sur les systèmes temps réel à criticité mixte (e.g. [BBD11]).

Ces systèmes ont été proposés pour répondre aux besoins des industriels de faire fonctionner différents types d'applications sur le même composant matériel. Ce besoin est apparu du fait que les possibilités matérielles ont fortement augmenté et que la consommation énergétique est devenue un enjeu primordial. En effet, les processeurs possèdent maintenant plusieurs processeurs et dans le futur seront sans doute exclusivement multiprocesseurs. Il est donc important de pouvoir tirer parti de ces processeurs, et l'idée de n'utiliser qu'une application par processeur rendant le processeur sous-utilisé n'est pas possible au niveau énergétique, même en exploitant au mieux les algorithmes permettant de réduire la consommation énergétique de ces systèmes.

		Sévérité			
		Négligeable	Marginale	Critique	Catastrophique
Fréquent	$> 10^{-3}$	Indésirable	Inacceptable	Inacceptable	Inacceptable
Probable	10^{-3} à 10^{-4}	Tolérable	Indésirable	Inacceptable	Inacceptable
Occasionnel	10^{-4} à 10^{-5}	Tolérable	Tolérable	Indésirable	Inacceptable
Éloigné	10^{-5} à 10^{-6}	Acceptable	Tolérable	Tolérable	Indésirable
Improbable	10^{-6} à 10^{-7}	Acceptable	Acceptable	Tolérable	Tolérable
Invraisemblable	$\leq 10^{-7}$	Acceptable	Acceptable	Acceptable	Acceptable

Table 6.1 – Matrice des risques *IEC 61508* [IEC].

Il existe un certain nombre de standards relatifs au taux de défaillance des systèmes et leur criticité. La matrice des risques pour le standard *IEC 61508* est par exemple détaillée dans le Tableau 6.1. Cette matrice donne les niveaux maximum de défaillances autorisées en fonction de la criticité de l'application. Par exemple, la norme *IEC 61508* considère qu'il est tolérable pour une tâche avec une sévérité marginale d'avoir un défaut selon une probabilité comprise entre 10^{-4} et 10^{-5}.

La modélisation proposée par Vestal [Ves07] définit un WCET pour chaque niveau de criticité, pour représenter par exemple les différents niveaux d'exigence des autorités de certification. Dans cette représentation, plus le niveau de criticité est élevé, plus le WCET est important. Si l'on suppose deux niveaux de criticité, le système est par défaut dans le niveau de criticité le plus faible et lorsqu'une tâche dépasse le WCET de son niveau de criticité faible, le système passe alors en mode haute criticité où les objectifs sont différents. Lorsque le système se trouve dans un niveau de criticité donné, les solutions existantes font généralement l'hypothèse que les échéances de toutes les tâches dont la criticité est plus faible que le niveau de criticité actuel ne sont alors plus garanties, et ce pour permettre de garantir les échéances des tâches à plus forte priorité. Cette modélisation se base sur le fait que les tâches à haute

criticité ne passeront que rarement dans un état de criticité élevé. Dans le cas contraire, cela signifie que les tâches à faible criticité seront ignorées et donc pourront potentiellement manquer régulièrement leurs échéances.

Nous avons choisi une modélisation différente de celle de Vestal en gardant néanmoins l'idée de ne pas garantir le WCET de toutes les tâches à faible criticité. Nous pensons en effet qu'avoir deux niveaux de fonctionnement pour les deux niveaux de criticité n'est pas nécessaire. Nous pourrions dans ce cas résoudre deux programmes linéaires différents pour chaque niveau de criticité en prenant ou non les tâches de faible criticité en considération. Nous avons choisi à la place de faire un pari sur le temps d'exécution des tâches à faible criticité.

Plus la criticité d'une tâche est faible, plus la tâche en question tolère des dépassements d'échéances, le nombre de dépassements d'échéances autorisés dépendant de la criticité de la tâche. C'est sur cette particularité que nous nous appuyons pour réduire la consommation énergétique de ces systèmes.

LPDPM-MC est un algorithme d'ordonnancement divisé en deux parties hors et en-ligne utilisant l'approche du chapitre 4. Hors-ligne, un ordonnancement est généré dans une hyperpériode divisée en intervalles. La programmation linéaire est utilisée pour calculer le poids de chaque tâche sur chaque intervalle. Dans le reste de cette section, nous présentons les modifications apportées aux contraintes temps réel pour profiter de la criticité mixte puis nous introduisons la fonction objectif pour minimiser la consommation énergétique de ces systèmes. FPZL est utilisé pour ordonnancer les tâches à l'intérieur des intervalles. Nous verrons en fin de section comment ordonnancer les intervalles pour obtenir le meilleur compromis entre efficacité énergétique et ordonnançabilité des tâches à faible criticité.

6.1.2 Adaptation des contraintes et fonction objectif

Nous avons adapté le programme linéaire (PL 3) présenté au chapitre 4 pour les systèmes temps réel à criticité mixte. Nous détaillons dans cette section les modifications des contraintes temps réel et de la fonction objectif.

Nous avons utilisé le programme linéaire de LPDPM2 et non celui de LPDPM1 malgré le fait que nos simulations montrent que LPDPM1 obtient de meilleurs résultats. Nous pensons que ce choix se justifie par le fait LPDPM2 permet d'exprimer de manière correcte le problème en calculant la taille et la consommation énergétique de chaque période d'inactivité. Au contraire, la modélisation utilisée dans le programme linéaire (PL 2) de LPDPM1 n'est qu'une simplification qui consiste à minimiser le nombre de périodes d'inactivité.

Nous définissons trois termes spécifiques aux systèmes à criticité mixte. L'utilisation globale des tâches à faible criticité est notée U_{LO} tandis que celle des tâches à haute criticité est U_{HI}. U_{LO} est égale à la somme des utilisations des tâches à faible criticité et U_{HI} est égale à celle correspondant aux tâches à haute criticité.

Le paramètre α, un nombre réel compris entre 0 et 1, permet de contrôler le temps processeur qui est alloué aux tâches à faible criticité. C'est le pourcentage du WCET qui est réservé pour chaque tâche à faible criticité. Une valeur de α faible permet de réduire de façon significative la consommation énergétique car peu de temps est alloué aux tâches à faible criticité et le temps d'inactivité des processeurs est plus large. Il permet ainsi d'utiliser

davantage les états basse-consommation. Au contraire, une valeur de α trop importante ne permet pas de réaliser d'économie d'énergie. Enfin, $\alpha = 1$ permet de générer le même ordonnancement que pour les systèmes temps réel dur.

6.1.2.1 Modification des contraintes temps réel

Nous rappelons ici brièvement le vocabulaire utilisé dans la suite de cette section. L'hyperpériode est divisée en intervalles et la taille de l'intervalle k est $|I_k|$. Chaque travail j a un WCET égal à $j.c$ et $w_{j,k}$ correspond au poids du travail j sur l'intervalle k. Le programme linéaire permet de calculer le poids de chaque tâche sur chaque intervalle.

Les contraintes du programme linéaire (PL 1) présenté en section 4.2.2 restent valides pour les tâches à haute criticité qui ne doivent pas manquer d'échéances. Chacune de ces tâches reste donc soumise aux contraintes (4.6), (4.7) et (4.8).

Comme nous autorisons les dépassements d'échéances pour les tâches à faible criticité, celles-ci sont également soumises aux contraintes (4.6) et (4.7). Cependant, nous ne réservons pas le WCET des tâches à faible criticité mais seulement un pourcentage. Comme vu au chapitre 4, le paramètre α correspond à ce pourcentage, avec $0 \leq \alpha \leq 1$. La contrainte (4.8) devient :

$$\forall j_{\mathrm{LO}}, \quad \sum_{k \in E_j} w_{j,k} \times |I_k| \geq \alpha \times j.c \tag{6.1}$$

où j_{LO} représente un travail de faible criticité. De même, j_{HI} représente un travail de haute criticité.

Pour chaque travail à faible criticité j_i, $\alpha \times j.c$ est le temps d'exécution minimal qui doit être réservé hors-ligne. α indique le pourcentage du WCET qui est réservé pour chaque travail. Mais comme la valeur de α est unique pour toutes les tâches de faible criticité, cela signifie qu'aucune échéance pour ces tâches n'est garantie. Cette affirmation n'est pas problématique en pratique car les tâches n'utilisent pas leur WCET lors de l'exécution. Néanmoins, si le travail manque son échéance, nous faisons l'hypothèse que le travail est abandonné, cette hypothèse est abordée plus en détails en section 6.1.3.1.

Valeur de α. La valeur de α est choisie par le concepteur du système suivant son objectif, qui peut être de diminuer la consommation énergétique du système ou de réduire le risque d'avoir des dépassements d'échéances pour les tâches à faible criticité. Une valeur de α petite va augmenter le risque qu'un dépassement d'échéances survienne tandis qu'une valeur de α importante ne permettra pas de réduire la consommation énergétique de manière significative. Pour faciliter cette décision, il est possible d'utiliser la distribution des temps d'exécution des tâches à faible criticité. Par exemple, α peut être choisi de telle sorte que la probabilité que l'AET de la tâche en question soit plus important que $\alpha \times j.c$, soit en dessous d'un certain pourcentage.

Pour illustrer le choix de la valeur α, nous avons utilisé la loi de Gumbel avec les paramètres $\mu = 2$ et $\beta = 4$ pour les valeurs des AET en comparaison avec le WCET, comme suggéré par Cucu-Grosjean et al. [CGSH+12]. La Figure 6.1 présente la densité de probabilité (à

gauche) et la fonction de répartition (à droite) de cette loi. Dans cet exemple, la probabilité que le temps d'exécution soit plus important que $0.6 \times WCET$ est de 88 %. En s'aidant de cette information, le concepteur du système peut choisir la valeur de α qu'il estime la plus appropriée.

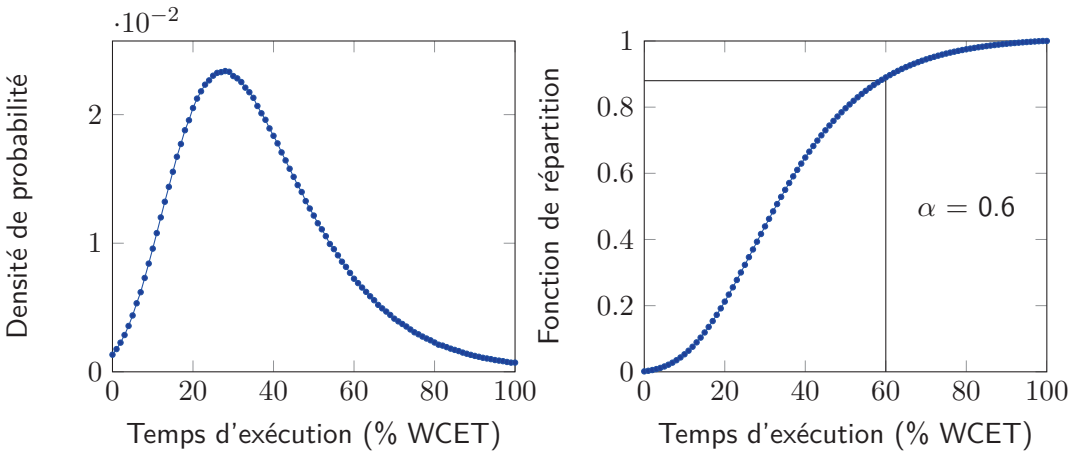

FIGURE 6.1 – Distribution des AET des tâches à faible criticité (Loi de Gumbel).

Diminution du nombre de processeurs. Comme le temps d'exécution réservé pour les tâches à faible criticité est inférieur à leur WCET, le temps d'exécution total de l'ensemble de tâches diminue. Le temps d'exécution de la tâche τ' doit par conséquent être augmenté pour que l'utilisation globale de l'ensemble de tâches soit toujours égale au nombre de processeurs. Mais le temps d'exécution de τ' est au mieux égal à l'hyperpériode (i.e. l'utilisation de τ' ne peut dépasser 1). Si w_k est le poids de τ' sur l'intervalle k, cela correspond à :

$$\sum_k w_k \times |I_k| \leq H \tag{6.2}$$

Cependant, il peut y avoir des ensembles de tâches et des valeurs de α pour lesquels l'utilisation de τ' doit dépasser 1 pour que l'utilisation reste égale à m. Le paramètre α est donc soumis à l'équation suivante pour que l'utilisation suivante pour que l'utilisation de τ' reste inférieure ou égale à 1 :

$$U_{\text{HI}} + \alpha \times U_{\text{LO}} \geq m - 1 \tag{6.3}$$

Il est possible de contourner ce problème en supprimant un processeur lorsque $m - 2 < U_{\text{HI}} + \alpha \times U_{\text{LO}} < m - 1$. Dans ce cas, l'utilisation de τ' reste inférieure à 1 et tous les calculs se font ensuite sur $m - 1$ processeurs, de même que l'exécution du système en-ligne. C'est l'hypothèse que nous faisons pour la suite de ce chapitre. Suivant la valeur de α choisie, nous supprimons du système un ou plusieurs processeurs si possible. L'état basse-consommation le plus économe en énergie est alors activé sur ces processeurs pour toute la durée de l'exécution. Toute les valeurs de α sont donc possibles.

L'utilisation globale du système de tâches est maintenant comprise entre $m-1$ et m, l'équation (6.1) peut donc être réécrite ainsi :

$$\forall j_{\text{LO}},\ \sum_{k \in E_j} w_{j,k} \times |I_k| = \alpha \times j.c \tag{6.4}$$

En effet, pour réduire la consommation énergétique, il sera toujours plus profitable de réserver le temps processeur minimal autorisé aux tâches à faible criticité pour augmenter au maximum la taille des périodes d'inactivité.

6.1.2.2 Fonction objectif du programme linéaire

Cette sous-section détaille la fonction objectif du programme linéaire pour minimiser la consommation énergétique du système. Nous utilisons une fonction objectif similaire à celle utilisée au chapitre précédent pour LPDPM car l'objectif est toujours de minimiser la consommation énergétique. Nous reprenons donc les notations introduites au chapitre précédent en section 5.2.3 qui sont résumées dans le Tableau 5.4. De façon similaire, la tâche τ' est divisée en deux sous-tâches τ'_b et τ'_e qui sont exécutées en début et fin d'intervalles. Il y a une seule période d'inactivité par intervalle et nous pouvons calculer la consommation énergétique P_k en suivant la méthode détaillée en section 5.2.3. La fonction objectif de notre programme linéaire est donc :

$$\text{Minimiser} \sum_k P_k \tag{6.5}$$

Nous avons maintenant notre programme linéaire en variables mixtes complet (PL 3). qui permet de calculer le poids de chaque tâche sur chaque intervalle. La section suivante propose ensuite des algorithmes pour ordonnancer les tâches à l'intérieur des intervalles pour réduire la consommation énergétique et limiter le nombre de violations d'échéances des tâches de plus faible criticité.

$$(PL\ 3) \qquad \begin{cases} \text{Minimiser} \sum_k P_k \\ \forall k,\ \sum_{j \in J_k} w_{j,k} \leq m \\ \forall k, \forall j,\ 0 \leq w_{j,k} \leq 1 \\ \forall j_{\text{HI}},\ \sum_{k \in E_j} w_{j,k} \times |I_k| = j.c \\ \forall j_{\text{LO}},\ \sum_{k \in E_j} w_{j,k} \times |I_k| = \alpha \times j.c \\ P_k = \sum_s LP_{s,k}(Cons_s \times q_k + Pen_s) \end{cases}$$

6.1.3 Ordonnancement en-ligne

Pour ordonnancer les tâches à l'intérieur des intervalles en utilisant les poids calculés hors-ligne, nous utilisons FPZL comme détaillé au chapitre 4. La tâche τ' est toujours divisée en deux sous-tâches qui sont exécutées en début et fin d'intervalle et ont donc par conséquent respectivement la plus forte et la plus faible priorité.

Si toutes les tâches utilisent leur WCET, les échéances de toutes les tâches à faible criticité seront systématiquement violées. Mais lors de l'exécution, les tâches peuvent ne pas consommer la totalité de leur WCET. Elles libèrent donc du *slack time* qui correspond à la différence entre le WCET d'une tâche et son AET. Nous voulons utiliser ce *slack time* pour améliorer l'ordonnançabilité des tâches à faible criticité en leur donnant plus de temps pour s'exécuter et également réduire la consommation énergétique. La solution détaillée dans cette section se rapproche de celle du chapitre précédent pour des systèmes temps réel dur.

Hors-ligne, nous n'avons réservé qu'un pourcentage du WCET des tâches à faible criticité. La somme de tous les poids n'est donc plus égale au WCET du travail. Soit t_j cette différence :

$$t_j = j.c - \sum_{E_j} w_{j,k} \tag{6.6}$$

Un travail d'une tâche à faible criticité va manquer son échéance si son temps d'exécution est plus important que $j.c - t_j$. La valeur de t_j dépend de la valeur de α qui a été fixée plus haut. Nous utilisons t_j pour savoir si une tâche à faible criticité peut être exécutée en fonction du *slack time* disponible. Plusieurs solutions sont envisageables, nous avons choisie une solution agressive qui privilégie la réduction de la consommation énergétique à l'ordonnançabilité des tâches de faible criticité. Nous avons fait ce choix car augmenter la taille des périodes d'inactivité permet de réduire la consommation énergétique de façon certaine tandis que donner le *slack time* aux tâches à faible criticité ne garantit pas que les échéances soient respectées. L'algorithme est détaillé ci-dessous :

1. Si le poids de τ' sur l'intervalle courant peut être augmenté (i.e. sa laxité est strictement supérieure à 0), le *slack time* est donné à τ' pour augmenter la taille de la période d'inactivité courante.
2. S'il existe au minimum un travail à faible criticité avec $t_j > 0$, le *slack time* est utilisé pour exécuter les tâches à faible criticité où $t_j > 0$. t_j est mis à jour lors de l'exécution et l'exécution se termine lorsque $t_j = 0$.
3. Sinon, un état basse-consommation est activé suivant la taille de la période d'inactivité.

Nous illustrons les scénarios possibles avec l'ensemble de tâches du tableau 6.2 composé 3 tâches ordonnancées sur deux processeurs. L'hyperpériode est ici de 12 et est découpée en trois intervalles de taille 3. Les AET de τ_1 et τ_2 sont respectivement de 4 et 7 tandis que les AET des 3 travaux de τ_3 sont 2, 2 et 1. Nous supposons $\alpha = \frac{2}{3}$. Nous réservons donc pour les tâches τ_2 et τ_3 des temps d'exécution respectivement de 6 et 2.

Les temps d'exécution réels des travaux dans les intervalles sont représentés dans le Tableau 6.3. Pour chaque tâche de l'ensemble de tâches Γ, la première colonne représente le

poids tel qu'obtenu en résolvant le programme linéaire (i.e. WCET). La deuxième colonne représente le temps processeur réellement utilisé lors de l'exécution (i.e. AET).

	τ_1	τ_2	τ_3
WCET	7	9	3
$\alpha\times$ WCET	-	6	2
Période	12	12	4
Criticité	HIGH	LOW	LOW

TABLE 6.2 – Ensemble de tâches ordonnancé en Figures 6.2 et 6.3.

	τ'_b	τ'_e	τ_1		τ_2		τ_3	
			WCET	AET	WCET	AET	WCET	AET
I_1	1	0	3	3	2	2	2	2
I_2	0	2	2	1	2	2	2	2
I_3	2	2	2	0	2	3	2	1

TABLE 6.3 – Temps d'exécution de l'ensemble de tâches des Figures 6.2 et 6.3.

Sur la Figure 6.2, les tâches utilisent leur WCET. Donc tous les travaux de τ_2 et τ_3 violent leur échéance. Sur la Figure 6.3, les tâches utilisent leur AET. Donc à $t = 5$, lorsque τ_1 termine son exécution, le *slack time* est cédé à τ', ce qui correspond ci-dessus au scénario 1.

La tâche τ_1 termine son exécution dans l'intervalle I_2 et n'est pas exécutée dans l'intervalle I_3. Le *slack time* est donc cédé à τ'. Dans cet intervalles les tâches τ_2 et τ_3 utilisent leur temps d'exécution réservé jusqu'à l'instant $t = 11$ où τ_3 termine son exécution. Le *slack time* libéré est donné à τ_2 car τ' est déjà en cours d'exécution sur le premier processeur, ce qui correspond ci-dessus au scénario 2. Sur cet exemple, aucune échéance n'est violée.

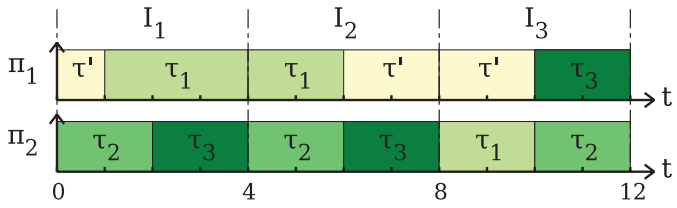

FIGURE 6.2 – Ordonnancement de l'ensemble de tâches du Tableau 6.2 en utilisant le WCET.

D'autres solutions plus complexes sont envisageables à la place de l'algorithme présenté ci-dessous. Lorsque du *slack time* est libéré et qu'il n'est plus possible de le donner à τ', le choix de la tâche à faible criticité à exécuter peut être :

— Choisir le travail ayant le plus faible t_j, c'est-à-dire le travail ayant le temps d'exécution hors exécution réservé le plus important, soit le travail ayant le plus de chances de finir son exécution.

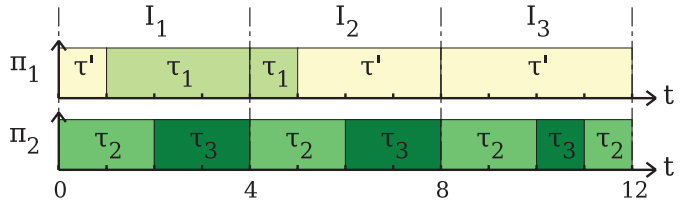

Figure 6.3 – Ordonnancement de l'ensemble de tâches du Tableau 6.2 en utilisant l'AET.

— Choisir le travail ayant l'échéance la plus rapprochée.

— Choisir le travail ayant la priorité la plus importante en ayant au préalable donné une priorité à chaque tâche de faible priorité pour savoir quelle tâche privilégier.

6.1.3.1 Dépassements d'échéances

Lorsqu'une tâche à faible criticité n'a plus de temps d'exécution disponible alors que son exécution n'est pas terminée, plusieurs solutions sont envisageables pour le système. La solution la plus simple est d'arrêter l'exécution de cette tâche qui ne pourra alors pas terminer son exécution. D'autres solutions moins catégoriques sont cependant également possibles. Si le système possède plus de deux niveaux de criticité, la tâche peut continuer son exécution si une autre tâche de plus faible criticité est prévue à l'exécution sur la suite du système. Le système donne alors la priorité à la tâche la plus critique. Cette solution n'est pas envisageable avec une tâche de plus haute criticité qui est prévue pour terminer son exécution. La tâche de faible criticité qui n'a pas terminé son exécution peut également être préemptée. Elle pourra ensuite être réexécutée si une tâche termine son exécution avant la date prévue.

6.2 Évaluation

Nous utilisons l'environnement de simulation du chapitre 5 pour évaluer LPDPM-MC. Des ensembles de tâches sont générés aléatoirement suivant le même procédé et sont ensuite ordonnancés sur des systèmes à 2 et 4 processeurs sur une durée égale à deux fois l'hyperpériode du système de tâches. Le processeur utilisé pour cette simulation est identique à celui utilisé au chapitre précédent pour évaluer LPDPM, ses 3 états basse-consommation ont été détaillés dans le Tableau 5.6 en page 70.

Pour chaque utilisation globale, 200 ensembles de tâches sont générés. Chaque ensemble de tâches comprend respectivement 5 et 10 tâches suivant le nombre de processeurs utilisés, avec respectivement 2 et 4 tâches à haute criticité dans ces ensembles de tâches. Nous avons choisi d'avoir moins de tâches à haute criticité que de tâches à faible criticité car nous pensons que cette proportion est la plus courante dans le monde industriel. Nous discuterons en fin de chapitre de l'évolution des résultats si cette proportion change.

Lors de cette simulation, chaque tâche à haute criticité consomme toujours son WCET tandis que les tâches à faible criticité ont un AET qui suit la loi de Gumbel comme présenté en section 6.1.2.1. Notre approche est aussi sollicité du fait que les tâches à faible criticité ne

consomment pas la totalité de leur WCET lors de l'exécution. Il serait également possible d'utiliser des temps d'exécution strictement inférieurs au WCET pour les tâches à haute criticité mais cela ne permettrait pas d'illustrer notre solution. Nous avons néanmoins tenu compte de cette possibilité dans notre algorithme d'ordonnancement présenté ci-dessus.

L'objectif de cette simulation est d'évaluer la consommation énergétique et le pourcentage de dépassements d'échéances suivant la valeur du paramètre α. Nous prenons quatre valeurs de α : 0.2, 0.4, 0.6 et 0.8. $\alpha = 1$ correspond à LPDPM, l'algorithme d'ordonnancement du chapitre précédent où toutes les tâches ont une haute criticité.

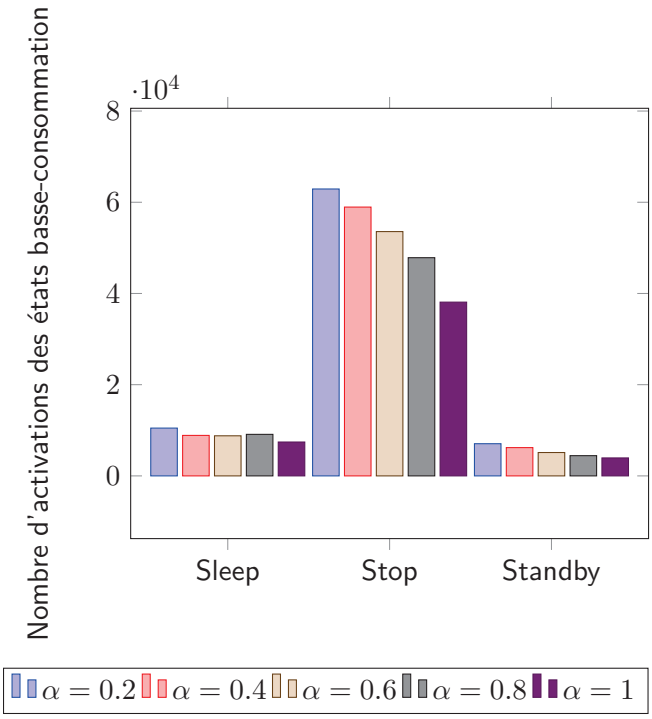

Figure 6.4 – Utilisation des états basse-consommation.

6.2.1 Utilisation des état basse-consommation

La Figure 6.4 représente l'utilisation des états basse-consommation pour les différentes valeurs de α testées. L'état basse-consommation *Stop* est de loin le plus utilisé. De plus la valeur de α est importante, plus les états basse-consommation les moins économes sont utilisés. Ce résultat est conforme aux résultats attendus car une valeur de α plus faible doit permettre d'économiser davantage d'énergie en créant des périodes d'inactivité plus étendues pour utiliser les états basse-consommation les plus efficaces.

6.2.2 Consommation énergétique

La Figure 6.5 représente la consommation énergétique de LPDPM-MC suivant 4 valeurs différentes de α pour 2 et 4 processeurs. Toutes les consommations énergétiques sont relatives à la consommation énergétique de LPDPM qui est toujours de 1. Contrairement au chapitre précédent, la consommation énergétique inclut en plus de la consommation énergétique lorsque le processeur est inactif la consommation énergétique lorsque les tâches non-critiques sont exécutées. En effet, le temps d'exécution de ces tâches n'est pas égal pour tous les algorithmes d'ordonnancement car ces tâches peuvent manquer des échéances, ce qui n'est pas possible avec les tâches à haute criticité.

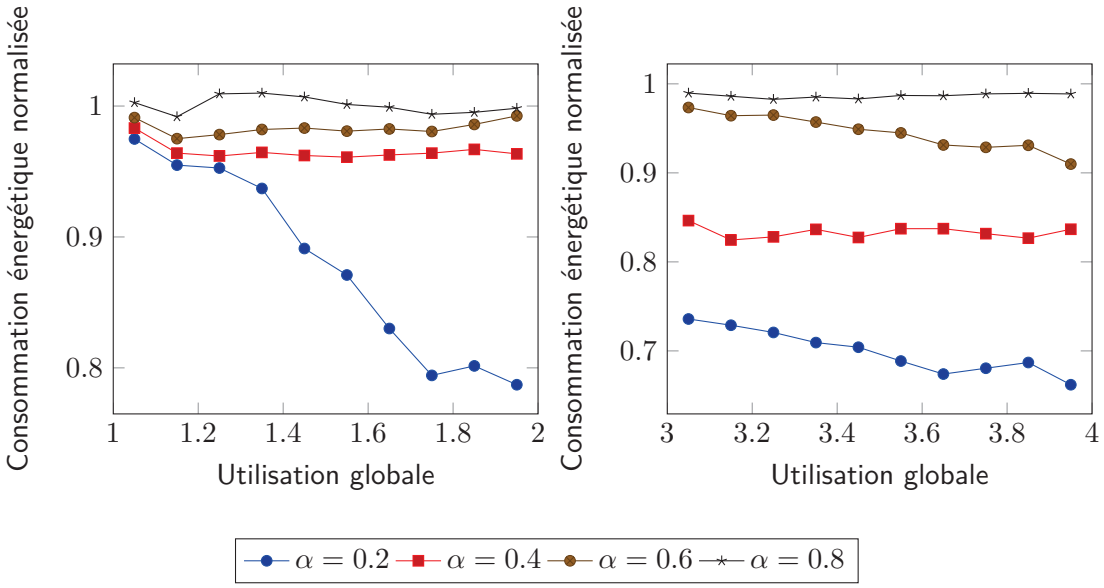

Figure 6.5 – Consommation énergétique pour 4 valeurs de α avec 2 (a) et 4 (b) processeurs.

Comme attendu, LPDPM-MC avec une valeur de α égale à 0.2 est la solution la plus efficace énergétiquement avec une réduction de la consommation énergétique allant jusqu'à 30%. Quand la valeur de α augmente, l'efficacité énergétique diminue mais est néanmoins de 10% pour $\alpha = 0.4$ et jusqu'à 8% pour $\alpha = 0.6$. Pour $\alpha = 0.8$, le gain est inférieur à 5%.

Les différences entre les consommations énergétiques des solutions présentées ici est moindre par rapport aux différences entre les algorithmes d'ordonnancement du chapitre précédent. Cette différence s'explique par le fait que la consommation énergétique inclut également la consommation énergétique des tâches de faible criticité. Or cette consommation énergétique est importante par rapport à la consommation énergétique lorsque les processeurs sont dans des états basse-consommation.

6.2.3 Dépassements d'échéances

La Figure 6.6 détaille pour 2 et 4 processeurs le pourcentage de dépassements d'échéances, soit le ratio entre le nombre de dépassements d'échéances le nombre total de travaux pour les

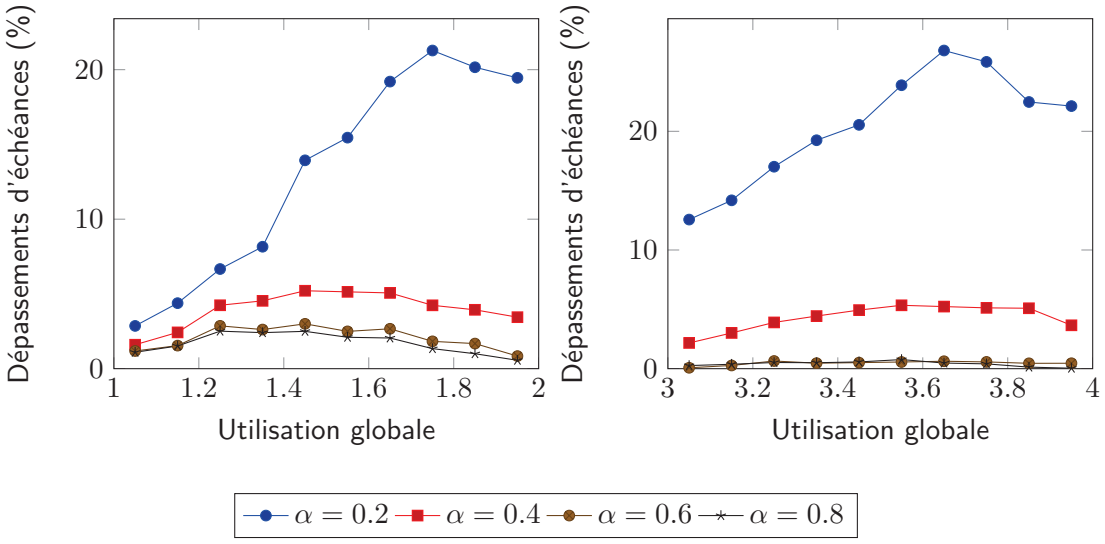

Figure 6.6 – Pourcentage de dépassements d'échéances pour 4 valeurs de α avec 2 (a) et 4 (b) processeurs.

tâches à faible criticité seulement. Les échéances des tâches à haute criticité ne peuvent elles pas être violées. Le nombre de dépassements d'échéances est plus élevé lorsque α est faible, avec un pourcentage compris entre 10 et 28% pour $\alpha = 0.2$. Quand la valeur d'α augmente, le pourcentage diminue et peut être considéré négligeable pour $\alpha > 0.4$.

Ces résultats sont conformes à ce qui pouvait être attendu au vu des valeurs de α et de la distribution choisie pour les valeurs des AET des tâches. En effet, pour $\alpha \leq 0.2$, la probabilité que le temps d'exécution réservé pour les tâches à faible criticité soit inférieur à l'AET est inférieure à 20 %.

Si la proportion de tâches à haute criticité augmente, le nombre de dépassements d'échéances risque d'augmenter pour les tâches à faible criticité car nous avons fait l'hypothèse que les tâches à haute criticité utilisent leur WCET lors de l'exécution. Il y aura donc moins de *slack time* disponible pour réduire le nombre de dépassements d'échéances.

Dans le cadre de nos simulations, nous avons affaire à des taux de défaillance plus importants que ceux définis par exemple dans la matrice de risque *IEC 61508* présentée en début de chapitre. Pour $\alpha = 0.8$, le risque est compris entre 10^0 et 10^{-2} ce qui correspond à un risque fréquent. Cette différence entre les taux de défaillance observés dans les simulations et ceux proposés par la norme *IEC 61508* peut s'expliquer par le fait que nos simulations ne sont faites que sur deux hyperpériodes. Le nombre de travaux sur cet intervalle de temps est donc limité. Pour obtenir des taux de défaillance inférieurs à 10^{-3}, nous envisageons de prendre des valeurs de α plus importantes et d'augmenter la durée de simulation.

Un autre point permettant d'expliquer les taux de défaillance importants est le fait que nous utilisons le WCET des tâches à haute criticité. Cette hypothèse permet de déterminer un gain minimun en consommation énergétique. Le gain obtenu en pratique sera toujours supérieur aux valeurs obtenues dans cette simulation si le temps d'exécution des tâches à

haute criticité est inférieur à leur WCET. Par ailleurs, le *slack time* libéré par une tâche à haute criticité est à priori plus important que celui libéré par une tâche à faible criticité car la différence entre l'AET et le WCET doit être plus important pour les tâches à haute criticité pour qu'aucun dépassement d'échéances n'ait lieu. Une perspective serait par conséquent d'effectuer des simulations en utilisant pour les tâches à haute criticité des temps d'exécution différents de leur WCET.

6.3 Conclusion

Cette section a présenté LPDPM-MC, un algorithme d'ordonnancement temps réel multiprocesseur permettant de réduire la consommation énergétique des systèmes temps réel à criticité mixte. Basé sur une approche similaire à LPDPM, il tire parti du fait que les tâches à plus faible criticité peuvent tolérer des dépassements d'échéances pour engendrer de larges périodes d'inactivité quitte à ce que les échéances de ces tâches ne soient pas assurées hors-ligne. Cette approche exploite le fait que les tâches ne consomment généralement pas leur WCET lors de l'exécution.

Hors-ligne, nous réservons seulement un pourcentage du WCET des tâches à faible criticité, ce pourcentage pouvant être choisi suivant les objectifs du concepteur du système. Plus le temps d'exécution réservé pour ces tâches à faible criticité est important, moins la probabilité d'avoir un dépassement d'échéances est important. Au contraire, réduire ce pourcentage augmente le risque d'avoir une échéance violée mais permet d'être plus agressif sur la réduction de la consommation énergétique en créant de plus larges périodes d'inactivité.

Lors de l'exécution, les tâches terminent souvent leur exécution plus tôt que prévu et libèrent du slack time. Nous avons donc proposé un algorithme en-ligne pour augmenter la taille des périodes d'inactivité existantes et éviter des dépassements d'échéances pour les tâches à faible criticité en utilisant ce slack time.

Nous avons présenté des évaluations montrant que suivant le niveau de criticité choisi, la consommation énergétique est réduite au détriment du nombre de dépassements d'échéances pour les tâches de faible criticité. Au maximum, nous obtenons une consommation énergétique réduite jusqu'à 30% pour 28% d'échéances violées. Un pourcentage de dépassements d'échéances réduite signifie une efficacité énergétique moindre, par exemple une consommation énergétique réduite 10% équivaut à 5% des échéances violées. Les résultats montrent également que la distribution choisie pour la valeur des AET des tâches influe sur les résultats obtenus. Il est également important de connaître les criticités des tâches pour choisir une valeur de α appropriée suivant le taux de dépassements d'échéances autorisé par le niveau de criticité choisi.

CHAPITRE 7

Conclusion et perspectives

7.1 Conclusion

Réduire la consommation énergétique des systèmes temps réel embarqués est un problème de plus en plus important notamment pour augmenter leur autonomie, ces systèmes ayant principalement comme source d'énergie des batteries dont l'autonomie est limitée. Diminuer la consommation énergétique est également un objectif de manière générale. L'objectif de cette thèse est par conséquent de réduire la consommation énergétique des systèmes temps réel embarqués. Nous nous sommes concentrés sur les systèmes multiprocesseurs qui vont remplacer les systèmes monoprocesseurs même dans les systèmes embarqués. Nous nous sommes focalisés sur des systèmes avec un nombre de cœurs compris entre 2 et 8.

La consommation énergétique des processeurs, les principaux composants de ces systèmes, est divisée en consommation dynamique et consommation statique. La première dépend de l'activité du processeur tandis que la seconde est essentiellement due aux courants de fuite et est présente quelque soit la fréquence d'exécution du processeur. Les premiers travaux cherchant à réduire la consommation énergétique des systèmes temps réel avaient pour objectif de réduire la consommation dynamique uniquement en réduisant la fréquence du processeur. Cette approche était correcte car la consommation dynamique était alors prépondérante. Or, les avancées technologiques des derniers processeurs (e.g. miniaturisation, augmentation de la densité des circuits) une augmentation de la consommation énergétique au détriment de la consommation dynamique.

Dans le cadre de cette thèse, nous nous sommes donc attachés à réduire principalement la consommation statique en utilisant les états basse-consommation des processeurs. Dans un état basse-consommation, le processeur est inactif et les courants de fuite fortement réduits mais une pénalité énergétique et un délai de transition sont nécessaires pour revenir à l'état actif. Du point de vue du système, un état basse-consommation doit donc être activé avec précaution pour garantir que le processeur sera de retour à temps à l'état actif pour qu'aucune

échéance ne soit violée.

Nous avons proposé plusieurs algorithmes d'ordonnancement temps réel multiprocesseurs optimaux pour réduire la consommation énergétique des systèmes temps réel embarqués : LPDPM1, LPDPM2 et LPDPM-MC. Les deux premiers sont destinés aux systèmes temps réel dur tandis que le dernier cible les systèmes temps réel à criticité mixte. LPDPM1 et LPDPM2 sont à notre connaissance les premiers algorithmes d'ordonnancement multiprocesseurs optimaux pour des ensembles de tâches temps-réel synchrones périodiques permettant de réduire la consommation énergétique. Pour LPDPM1 et LPDPM2, aucun dépassement d'échéances n'est autorisé tandis que les systèmes temps réel à criticité mixte tolèrent que des échéances soient violées pour les tâches de plus faible criticité. Cette caractéristique des systèmes temps réel à criticité mixte permet à LPDPM-MC d'être plus agressif pour réduire la consommation énergétique en faisant un compromis entre la consommation énergétique et le nombre de violations d'échéances pour ces tâches à faible criticité.

Les algorithmes d'ordonnancement que nous avons proposés utilisent une même approche. Un ordonnancement permettant de réduire la consommation énergétique est tout d'abord généré hors-ligne en utilisant la programmation linéaire. Le programme linéaire utilisé est différent suivant la modélisation du système et l'objectif recherché. En-ligne, lors de l'exécution du système, l'algorithme d'ordonnancement ordonnance les tâches suivant les résultats hors-ligne et permet de réduire davantage la consommation en élargissant les périodes d'inactivité lorsque les tâches ne consomment pas tout leur WCET.

Les algorithmes d'ordonnancement LPDPM1 et LPDPM2 sont destinés aux systèmes temps réel dur et permettent respectivement de minimiser le nombre de périodes d'inactivité et de minimiser la consommation statique. L'avantage de LPDPM1 est de donner un résultat efficace quel que soit le processeur utilisé car il n'a pas besoin des caractéristiques des états basse-consommation. LPDPM2 minimise lui la consommation énergétique en calculant la consommation énergétique de toutes les périodes d'inactivité en fonction du processeur utilisé. Nous avons évalué à l'aide de simulations ces deux solutions entre elles et en le comparant aux algorithmes d'ordonnancement existants RUN et U-EDF. La consommation énergétique lorsque les processeurs sont inactifs est jusqu'à 10 fois inférieure pour LPDPM1 et LPDPM2 comparé aux deux algorithmes d'ordonnancement RUN et U-EDF.

LPDPM-MC est le premier algorithme d'ordonnancement permettant de réduire la consommation statique des systèmes temps réel à criticité mixte où les tâches peuvent être de différentes criticités [BD13]. Cet algorithme utilise la même approche que LPDPM1 et LPDPM2 et tire parti du fait que les tâches de plus faible criticité peuvent tolérer des dépassements d'échéances. Il propose un compromis entre des dépassements d'échéances pour les tâches à faible criticité et la consommation énergétique. Le fait d'avoir des tâches de faible criticité permet à l'algorithme d'être plus agressif hors-ligne en réservant un temps d'exécution inférieur au WCET de ces tâches à faible criticité. Tout en conservant un taux de défaillance inférieur à 5% pour les tâches à faible criticité, nos simulations montrent que la consommation énergétique lorsque les processeurs sont inactifs et exécutent les tâches à faible criticité est réduite jusqu'à 10%.

7.2 Perspectives

Nous détaillons les perspectives envisagées pour améliorer les algorithmes d'ordonnancement que nous avons proposés. Nous discutons ensuite des extensions possibles à plus long terme au niveau de la modélisation du système.

7.2.1 Implémentation

Une première perspective est l'implémentation des algorithmes d'ordonnancement proposés dans cette thèse dans un noyau existant. Une implémentation et un calcul réel de la consommation énergétique permettrait de valider nos solutions et leur efficacité énergétique de façon expérimentale.

7.2.2 Perspectives générales

Dans cette sous-section nous proposons plusieurs solutions pour améliorer les performances de nos solutions. Ces solutions sont valables pour LPDPM1, LPDPM2 et LPDPM-MC.

7.2.2.1 Exécutions retardées

Les algorithmes d'ordonnancement que nous proposons sont oisifs et retardent l'exécution des tâches pour optimiser la taille des périodes d'inactivité comme nous l'avons vu au chapitre 3. Mais retarder l'exécution des tâches implique que certaines tâches peuvent être ordonnancées de telle sorte qu'elles finissent leur exécution au niveau de leur échéance. Ceci peut être problématique d'un point de vue pratique, en cas d'erreur lors de l'exécution car il n'existe aucune marge de manœuvre, par exemple si le WCET d'une tâche n'a pas été correctement calculé.

Pour éviter d'ordonnancer un travail trop près de son échéance, il serait possible d'ajouter une contrainte à notre programme linéaire pour limiter le temps d'exécution des travaux dans le dernier intervalle où ils sont présents. Si un travail est présent sur plusieurs intervalles, le poids du travail dans le dernier intervalle où il est présent devrait par exemple être inférieur à un pourcentage de la taille de cet intervalle.

Plus généralement, si un travail est présent sur plusieurs intervalles, nous pourrions privilégier les intervalles proches du début de l'exécution par rapport aux intervalles proches de l'échéance du travail en question. Nous définirions par exemple pour chaque travail j une variable $\zeta_{j,k}$ pour chaque intervalle k, cette variable étant un nombre entier égal au numéro de l'intervalle. Si un travail est présent sur trois intervalles, la valeur de ces variables $\zeta_{j,k}$ seraient alors respectivement $\{1, 2, 3\}$. Pour chaque travail j, un objectif secondaire serait alors :

$$\text{Minimiser} \sum_{k} \zeta_{j,k} \times w_{j,k} \tag{7.1}$$

En-ligne, nous avons présenté une solution d'ordonnancement à l'intérieur des intervalles qui utilise FPZL de manière assez « naïve ». Pour que les travaux ne soient pas exécutés

juste avant leur échéance, les travaux se trouvant dans leur dernier intervalle pourraient se voir attribuer une priorité plus importante. Plus généralement, il serait possible de donner une priorité plus importante aux travaux dont l'échéance est la plus proche, quel que soit l'intervalle dans lequel cette échéance se trouve.

7.2.2.2 Sous-travaux

Notre approche est basée sur l'hypothèse que les tâches utilisent leur WCET. Dans le cas où cette hypothèse est fausse (i.e. à l'exécution), des périodes d'inactivité peuvent apparaître à l'intérieur des intervalles. Ce phénomène est du au fait que l'hyperpériode est découpée en intervalles, ce qui a pour effet de découper chaque travail en plusieurs sous-travaux.

Tout est fait pour que les périodes d'inactivité créées à l'intérieur des intervalles permettent d'agrandir les périodes d'inactivité existantes mais dans le cas où cela se révèle impossible, l'approche choisie aura tendance à créer plus de périodes d'inactivité que les algorithmes d'ordonnancement classiques. En effet, si une tâche τ supposée être ordonnancée sur trois intervalles successifs termine son exécution sur le premier intervalle, cela crée immédiatement trois périodes d'inactivité sur l'intervalle courant et les deux intervalles suivants. Au contraire, un algorithme d'ordonnancement classique ne créé qu'une seule période d'inactivité supplémentaire.

Pour illustrer cette affirmation, nous avons ordonnancé l'ensemble de tâches 4 du Tableau 7.1 sur la Figure 7.1 en utilisant l'approche décrite dans ce chapitre. Si les tâches qui sont ordonnancées en fin d'intervalle finissent leur exécution avant leur WCET des périodes d'inactivité seront crées à la fin de chaque intervalle, pouvant ainsi rendre caduques les efforts faits hors-ligne pour optimiser la taille des périodes d'inactivité. Par exemple, si τ_1 ne consomme pas tout son WCET dans les intervalles 1 et 3, deux périodes d'inactivité seront créées.

	τ_1	τ_2	τ_3
WCET	0.8	2.4	4
Période	4	4	6

TABLE 7.1 – Ensemble de tâches 4.

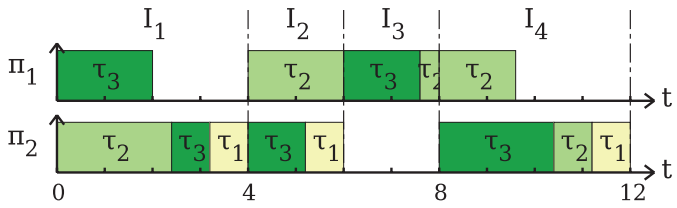

FIGURE 7.1 – Ordonnancement possible de l'ensemble de tâches 4.

Il est donc nécessaire de prévoir des solutions en-ligne pour éviter que le phénomène décrit ici n'apparaisse. Lorsqu'une tâche termine son exécution plus tôt que prévu, nous avons utilisé le *slack time* qu'elle libère pour augmenter la taille de la période d'inactivité courante.

Un point à noter est que les travaux sont en général présents sur plusieurs intervalles. Si un travail ne consomme pas son WCET, il va terminer son exécution dans un intervalle k et les intervalles suivants où son poids est supérieur à 0 pourront dès le début de l'intervalle donner ce poids à τ' pour augmenter la taille de la période d'inactivité. Le scénario dont nous avons discuté dans cette section ne peut donc se produire au maximum que sur un seul des intervalles où un travail est présent.

Une autre idée qui pourrait être exploitée est d'anticiper l'exécution des travaux des intervalles suivants. Si du *slack time* est libéré dans l'intervalle k, il serait possible de l'utiliser pour ordonnancer des tâches en utilisant leur poids dans l'intervalle $k+1$. Cela permettrait ainsi d'augmenter la taille de la période d'inactivité de l'intervalle $k+1$.

7.2.2.3 Suppression de l'hypothèse $m-1 < U < m$

La modélisation actuelle fait l'hypothèse que l'utilisation globale de l'ensemble de tâches est strictement comprise entre $m-1$ et m. Cette hypothèse est nécessaire pour notre solution mais d'autres évaluations pourraient être effectuées en la supprimant. En faisant varier l'utilisation globale entre 0 et m, il est toujours possible d'utiliser notre solution en disant que les $m - \lceil U \rceil$ processeurs inutiles sont toujours dans un état basse-consommation et ont une consommation énergétique par conséquent nulle.

Cette nouvelle modélisation serait profitable à notre solution car la consommation énergétique des processeurs supplémentaire serait nulle. Mais des algorithmes comme RUN ou U-EDF auraient également plus de processeurs disponibles pour ordonnancer les tâches, ce qui pourrait avoir comme conséquence de réduire le nombre de préemptions et migrations. Supprimer cette hypothèse permettrait également de faire des simulations avec des algorithmes non-optimaux de la littérature et d'éviter de faire des évaluations avec seulement U-EDF et RUN qui ne sont pas conçus pour maximiser la taille des périodes d'inactivité ou réduire la consommation énergétique.

Une autre approche pour ordonnancer les ensembles de tâches où $0 < U < m$ est d'ajouter une tâche τ' par processeur pour représenter l'inactivité de chaque processeur. Pour que l'utilisation globale du système soit toujours égale au nombre de processeurs après l'ajout de ces tâches τ', la somme de leur utilisation doit être égale à $m - U$ et l'utilisation individuelle de chaque tâche τ' doit être comprise entre 0 et 1. Ensuite, les deux solutions présentées au chapitre 5 peuvent être réutilisées suivant que l'on souhaite minimiser le nombre de périodes d'inactivité ou minimiser la consommation statique de chaque processeur. Les programmes linéaires peuvent en effet être réutilisés mais il est maintenant nécessaire de faire la démarche m fois, pour chacune des tâches τ'. L'inconvénient majeur de cette solution est par conséquent la complexité accrue du programme linéaire.

7.2.2.4 Préemptions et migrations

Pour diminuer le nombre de préemptions et de migrations générées par nos algorithmes d'ordonnancement, il serait possible d'étendre le programme linéaire en s'inspirant des travaux de Mégel et al. [Mé12, MSD10]. Leur algorithme d'ordonnancement utilise la même approche, mais dans l'objectif de réduire le nombre de préemptions et de migrations. Cet objectif est

important car l'un des principaux problèmes des algorithmes d'ordonnancement optimaux multiprocesseurs est le grand nombre de préemptions et de migrations.

Pour ce faire, nous devrions ajouter dans notre programme linéaire un mécanisme permettant de compter les préemptions et les migrations de chaque tâche, en suivant par exemple la méthode suivie par Mégel et al., et ajouter ensuite une fonction objectif permettant de réduire leur nombre. Nous aurions ainsi deux objectifs distincts, un système de poids pouvant alors être imaginé pour donner une importance supplémentaire à l'objectif choisi par le concepteur du système. Par exemple, soit PM_k une variable entière représentant le nombre de préemptions et de migrations dans l'intervalle k et α et β deux réels dans l'intervalle $[0, 1]$, la fonction objectif serait alors :

$$\text{Minimiser} \sum_k \alpha \times P_k + \beta \times PM_k \tag{7.2}$$

Résoudre le système linéaire initial avec ce nouvel objectif permettrait de réduire le nombre de préemptions du système tout en gardant une consommation énergétique réduite. La valeur des variables α et β est de la responsabilité du concepteur du système, suivant qu'il veut mettre l'accent sur la réduction de la consommation énergétique ou sur la réduction du nombre de préemptions et de migrations.

7.2.3 Systèmes temps réel à criticité mixte

Nous discutons dans cette section de plusieurs autres solutions possibles pour réduire spécifiquement la consommation énergétique des systèmes temps réel à criticité mixte.

7.2.3.1 Approche maximisant l'ordonnançabilité

Une autre solution est envisageable pour réduire la consommation statique en utilisant les états basse-consommation. Au lieu de minimiser la consommation énergétique en fixant une valeur pour α comme le fait LPDPM-MC, cette solution fixe une consommation énergétique à atteindre par rapport à la consommation énergétique lorsque $\alpha = 1$. La fonction objectif est alors de maximiser la valeur de α pour augmenter la probabilité que les tâches non critiques ne violent pas leur échéances.

LPDPM-MC permet de minimiser la consommation statique tout en garantissant le WCET des tâches à haute criticité et une partie seulement du WCET pour les tâches à faible criticité. À l'inverse, cette seconde approche, toujours en gardant l'idée de parier sur les temps d'exécution des tâches à faible criticité, calcule tout d'abord un ordonnancement en prenant l'hypothèse que toutes les tâches ont une haute criticité. La consommation énergétique de cet ordonnancement peut facilement être calculée, connaissant la durée et la consommation énergétique de toutes les périodes d'inactivité. Soit C cette consommation énergétique. L'idée est de fixer le gain énergétique souhaité par rapport à cette consommation énergétique de base avec $\alpha = 1$. Soit g le gain énergétique souhaité, avec g un nombre réel entre 0 et 1. La fonction objectif de cette solution est alors de maximiser la valeur de α tout en ayant comme contrainte que la consommation énergétique totale ne dépasse pas $g \times C$.

7.2.3.2 Gestion plus fine de l'ordonnançabilité

Dans le cadre de LPDPM-MC, la valeur de α est identique pour toutes les tâches du système, ce qui implique que toutes les tâches à faible criticité vont manquer leurs échéances si les tâches consomment leur WCET. Pour régler l'ordonnançabilité des tâches de manière plus fine, une variable binaire serait introduite pour chaque travail permettant de savoir si son échéance est garantie hors-ligne.

Formellement, nous définirions pour chaque travail j une variable binaire gd_j (*Guaranteed Deadline*) égale à 1 si l'échéance du travail j est garantie hors-ligne et 0 sinon :

$$gd_j = \begin{cases} 1 & \text{si } \sum_{k \in E_j} w_{j,k} \times |I_k| = j.c \\ 0 & \text{sinon} \end{cases} \tag{7.3}$$

La fonction objectif du programme linéaire resterait identique mais une contrainte sur le nombre d'échéances violées pourrait facilement être ajoutée. Par exemple :

$$\sum_j gd_j \geq \beta \times J \tag{7.4}$$

Où J est le nombre total de travaux dans l'hyperpériode et β ($\in [0, 1]$) est la proportion d'échéances qui ne doivent pas être violées. De même que pour la solution actuelle, les travaux dont l'échéance n'est pas garantie peuvent tout de même finir leur exécution en utilisant le *slack time* libéré lors de l'exécution.

Cette approche peut également être utilisée pour supporter plusieurs niveaux de criticité en utilisant une valeur de β différente pour chaque niveau de criticité. Il est alors possible de faire en sorte qu'un pourcentage des travaux aient leur échéance garantie, par exemple au niveau requis par un standard.

7.2.4 Modélisation

Nous dégageons dans cette section deux idées pour améliorer le modèle utilisé dans le cadre de cette thèse.

7.2.4.1 Tâches dépendantes

L'approche actuelle ne permet que d'ordonnancer des ensembles de tâches où toutes les tâches sont indépendantes. Or des applications nécessitent que certaines tâches soient par exemple exécutées avant d'autres. Nous pourrions alors étendre notre programme linéaire pour contraindre un travail à ne commencer son exécution que lorsque tous les travaux nécessaires à ce travail ont terminé leur exécution.

7.2.4.2 Consommation des périphériques

Nous avons choisi au chapitre 2 de nous concentrer sur la réduction de la consommation énergétique du processeur uniquement. Nous pensons que cette hypothèse est correcte avec les systèmes que nous ciblons mais la consommation énergétique des autres composants du

système pourrait également être considérée. Parmi ces composants, citons par exemple la mémoire ou les périphériques d'entrée/sortie. Ces périphériques proposent généralement des solutions au système pour réduire leur consommation énergétique, principalement la possibilité d'éteindre un périphérique lorsqu'il n'est pas en cours d'utilisation.

Dans le cas de la mémoire, il est également important de pouvoir choisir où allouer de la mémoire et de pouvoir déplacer des blocs de mémoires alloués lorsque l'on souhaite vider une partie de la mémoire pour pouvoir l'éteindre. Par exemple, les travaux de Ben Fradj [BF06] ont pour objectif la minimisation de l'énergie consommée dans des systèmes multiprocesseurs avec des mémoires multi-bancs. Dans l'architecture retenue, la mémoire n'est pas uniforme et est composée d'unités de différentes tailles dont la consommation peut varier. L'une des idées de la thèse est de répartir les données dans des mémoires appropriées pour permettre aux autres mémoires de rester inactives et ainsi économiser de l'énergie. La question à résoudre est la configuration des bancs mémoires, leur nombre, leur taille et l'allocation des bancs aux tâches du système.

Aydin et al [ADZ06] se sont également intéressés à la consommation énergétique des périphériques. Ils font l'hypothèse qu'un périphérique associé à une tâche doit rester actif durant toute l'exécution de la tâche. Ils cherchent alors à réduire la consommation dynamique du processeur en réduisant sa fréquence. Ils prennent cepdenant en compte la consommation statique des périphériques et essaient de l'optimiser en éteignant ces périphériques dès que possible.

Intégration à LPDPM. Pour pouvoir intégrer la gestion de la consommation énergétique des périphériques et de la mémoire à nos algorithmes d'ordonnancement, il est nécessaire de faire évoluer la modélisation actuelle. Nous définissons par exemple un ensemble P de périphériques et chaque tâche peut être associée ou non avec un ou plusieurs périphériques. Chaque périphérique a une consommation énergétique qui est constante lorsque le périphérique est actif et nulle lorsque celui-ci est au repos. Nous laissons de côté ici les problèmes qui peuvent intervenir si deux tâches utilisant le même périphérique sont actives en même temps, nous supposons alors qu'elles peuvent l'utiliser simultanément.

En prenant la même hypothèse que Aydin et al., lorsqu'une tâche est active, tous ses périphériques doivent également l'être. L'objectif est alors de réduire la consommation énergétique du système en intégrant et la consommation énergétique des processeurs et celle des périphérique. Pour chaque intervalle k, soit $P_{p,k}$ la consommation énergétique du périphérique p, la fonction objectif du programme linéaire pourrait être alors :

$$\text{Minimiser } \sum_k P_k + \sum_k \sum_p P_{p,k} \tag{7.5}$$

Il est également nécessaire d'ajouter des contraintes pour que les périphériques soient actifs dès que les tâches dépendants de ces périphériques le sont.

7.2.4.3 Couplage avec des modèles thermiques et de fiabilité

Un lien existe entre la température du système et sa consommation énergétique [CWT09, FCWT09, FCWT11, BA13], plus particulièrement la consommation statique qui dépend de

la température ambiante. Cette donnée n'est que rarement prise en compte dans les études sur la consommation énergétique, mais elle pourrait devenir importante dans les années à venir. Nous souhaiterions par conséquent intégrer ce lien dans notre modèle pour que la température et donc la consommation énergétique soit davantage réduite. À noter que le choix que nous avons fait de réduire en priorité la consommation statique plutôt que de la consommation dynamique est renforcé avec la prise en compte de la température.

Comme vu au chapitre 2, la consommation statique est principalement due aux courants de fuite. Nous l'avons définie comme étant constante ce qui est uniquement correct si la température n'est pas prise en compte. Avec la température, la puissance liée aux courants de fuite P_{leak} peut être modélisée ainsi [HBA03] :

$$P_{leak} = P_{leak110} e^{\beta * (T_c - 110)} \tag{7.6}$$

Où P_{leak} est le courant de fuite à la température de 100°C et T_c la température du processeur c. La consommation énergétique due aux courants de fuite augmente donc avec la température, il est donc nécessaire de réguler la température du système pour éviter des pics thermiques.

Nous proposons une première solution en-ligne aisément intégrable à nos solutions pour diminuer la température des processeurs en répartissant la charges entre tous les processeurs du système. Le processeur sur lequel est ordonnancé τ' est choisi selon FPZL lors de l'ordonnancement de la tâche τ'. Mais lorsque τ' est ordonnancée en début d'intervalle, il est possible de choisir explicitement le processeur sur lequel τ' est exécutée et où un état basse-consommation va pouvoir être activé. Nous avons choisi de changer à chaque exécution de τ' le processeur activant un état basse-consommation pour répartir la charge entre tous les processeurs. Cette approche permet de ne pas avoir de processeurs toujours actifs ce qui pourrait entrainer un hausse de la température.

Bibliographie

[A9] *ARM Cortex-A9 MPCore Technical Reference Manual*.

[AB04] J. H. Anderson and S. K. Baruah. Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms. In *Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 428–435, 2004.

[ABM+04] T. Austin, D. Blaauw, S. Mahlke, T. Mudge, C. Chakrabarti, and W. Wolf. Mobile supercomputers. *Computer*, 37(5) :81–83, May 2004.

[ADZ06] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 313–322, 2006.

[AHS05] J. Anderson, P. Holman, and A. Srinivasan. Fair scheduling of real-time tasks on multiprocessors. In *Handbook on Scheduling Algorithms, Methods, and Models*, 2005.

[AJ02] B. Andersson and J. Jonsson. Preemptive multiprocessor scheduling anomalies. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pages 12–, 2002.

[AMAMM01] H. Aydin, P. Mejía-Alvarez, D. Mossé, and R. Melhem. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 95–, 2001.

[AMCM03] N. AbouGhazaleh, D. Mossé, B. Childers, and R. Melhem. *Toward the placement of power management points in real-time applications*, pages 37–52. 2003.

[AP11] M. Awan and S. Petters. Enhanced race-to-halt : A leakage-aware energy management approach for dynamic priority systems. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems*, pages 92–101, 2011.

[AP13] M. Awan and S. Petters. Energy aware partitioning of tasks onto a heterogeneous multi-core platform. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013.

[ARI] ARINC. ARINC 653-1 Avionics Application Software Standard Interface.

[AY03] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pages 113.2–, 2003.

[AYP13] M. A. Awan, P. M. Yomsi, and S. 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*, pages 129–138, 2013.

[BA13] K. Baati and M. Auguin. Temperature-aware DVFS-DPM for real-time applications under variable ambient temperature. In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems*, pages 13–20, 2013.

[Bak03] T. P. Baker. Multiprocessor edf and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 120–, 2003.

[BB04] E. Bini and G. C. Buttazzo. Biasing effects in schedulability measures. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 196–203, 2004.

[BBA10a] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. Cache-related preemption and migration delays : Empirical approximation and impact on schedulability. In *Proceedings of the Sixth International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, 2010.

[BBA10b] A. Bastoni, B. B. Brandenburg, and J. H. Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 14–24, 2010.

[BBA10c] M. Bhatti, C. Belleudy, and M. Auguin. An inter-task real time DVFS scheme for multiprocessor embedded systems. In *Proceedings of the 2010 Conference on Design & Architectures for Signal & Image Processing*, pages 136–143, 2010.

[BBA11] M. K. Bhatti, C. Belleudy, and M. Auguin. Hybrid power management in real time embedded systems : an interplay of DVFS and DPM techniques. *Real-Time Systems*, 47 :143–162, March 2011.

[BBBM00] L. Benini, A. Bogliolo, R. Bogliolo, and G. D. Micheli. A survey of design techniques for system-level dynamic power management. *IEEE Transactions on VLSI Systems*, 8 :231–248, 2000.

[BBD+10] S. K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. In *Proceedings of the 35th international conference on Mathematical foundations of computer science*, pages 90–101, 2010.

[BBD11] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium*, pages 34–43, 2011.

[BBMB13] M. Bambagini, M. Bertogna, M. Marinoni, and G. Buttazzo. On the impact of runtime overhead on energy-aware scheduling. In *Workshop on Power, Energy, and Temperature Aware Real-time Systems*, 2013.

[BCPV93] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress : a notion of fairness in resource allocation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 345–354, 1993.

[BD13] A. Burns and R. Davis. Mixed criticality systems - a review. Technical report, 2013.

[Ber07] M. Bertogna. *Real-Time Scheduling Analysis for Multiprocessor Platforms*. PhD thesis, Scuola Superiore Sant'Anna, Pisa, 2007.

[BF06] H. Ben Fradj. *Optimisation de l'énergie dans une architecture mémoire multi-bancs pour des applications multi-tâches temps réel*. PhD thesis, Université de Nice Sophia-Antipolis, Dec 2006.

[BFBA09] M. Bhatti, M. Farooq, C. Belleudy, and M. Auguin. Controlling energy profile of rt multiprocessor systems by anticipating workload at runtime. In *SYMPosium en Architectures nouvelles de machines*, 2009.

[Bor99] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4) :23 –29, jul-aug 1999.

[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. *Journal of Real-Time Systems*, 2, October 1990.

[CG06] H. Cheng and S. Goddard. Online energy-aware i/o device scheduling for hard real-time systems. In *Proceedings of the conference on Design, automation and test in Europe*, pages 1055–1060, 2006.

[CGSH+12] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 91–101, 2012.

[CH10] A. Carroll and G. Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, pages 21–21, 2010.

[CHK06] J.-J. Chen, H.-R. Hsu, and T.-W. Kuo. Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In *Proceedings of the 12th IEEE Real-Time & Embedded Technology & Applications Symposium*, pages 408–417, 2006.

[CK06] J.-J. Chen and T.-W. Kuo. Procrastination for leakage-aware rate-monotonic scheduling on a dynamic voltage scaling processor. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, pages 153–162, 2006.

[CK07] J.-J. Chen and C.-F. Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 28–38, 2007.

[CPL] CPLEX : ILOG CPLEX 12.4 (http ://www.ilog.com).

[CWT09] J.-J. Chen, S. Wang, and L. Thiele. Proactive speed scheduling for real-time tasks under thermal constraints. In *Proceedings of the 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, pages 141–150, 2009.

[DA08a] V. Devadas and H. 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*, pages 99–108, 2008.

[DA08b] V. Devadas and H. Aydin. Real-time dynamic power management through device forbidden regions. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 34–44, 2008.

[DB11a] R. Davis and A. Burns. Fpzl schedulability analysis. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 245 –256, 2011.

[DB11b] R. I. Davis and A. Burns. Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Syst.*, 47(1) :1–40, January 2011.

[DB11c] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Compututing Surveys*, 43(4) :35 :1–35 :44, October 2011.

[DJ12] N. Dandrimont and M. Jan. Performance simulation of a DPM scheduling policy on realistic processors. In *IEEE Real-Time and Embedded Technology and Applications Symposium (Work in Progress session)*, 2012.

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

[DR06] G. Dhiman and T. S. Rosing. Dynamic power management using machine learning. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 747–754, 2006.

[Fau11] F. Fauberteau. *Sûreté temporelle pour les systèmes temps réel multiprocesseurs*. PhD thesis, Université Paris-Est, 2011.

[FCWT09] N. Fisher, J.-J. Chen, S. Wang, and L. Thiele. Thermal-aware global real-time scheduling on multicore systems. In *Proceedings of the 2009 15th IEEE Symposium on Real-Time and Embedded Technology and Applications*, pages 131–140, 2009.

[FCWT11] N. Fisher, J.-J. Chen, S. Wang, and L. Thiele. Thermal-aware global real-time scheduling and analysis on multicore systems. *Journal of Systems Architecture*, 57(5) :547 – 560, 2011.

[Fis07] N. Fisher. *The multiprocessor real-time scheduling of general task systems*. PhD thesis, University of North Carolina at Chapel Hill, 2007.

[GJ90] M. R. Garey and D. S. Johnson. *Computers and Intractability ; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[GN07] B. Gaujal and N. Navet. Dynamic voltage scaling under edf revisited. *Real-Time Systems*, 37 :77–97, October 2007.

[Gru02] F. Gruian. *Energy-Centric Scheduling for Real-Time Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 2002.

[HBA03] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 217–222, 2003.

[HCK06] C.-M. Hung, J.-J. Chen, and T.-W. Kuo. Energy-efficient real-time task scheduling for a DVS system with a Non-DVS processing element. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 303–312, 2006.

[HPS98] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 653–656, 1998.

[HSC+11] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo. Applying real-time interface and calculus for dynamic power management in hard real-time systems. *Real-Time Systems*, 47 :163–193, March 2011.

[HXW+10] H. Huang, F. Xia, J. Wang, S. Lei, and G. Wu. Leakage-aware reallocation for periodic real-time tasks on multicore processors. In *Proceedings of the 2010 Fifth International Conference on Frontier of Computer Science and Technology*, pages 85–91, 2010.

[IEC] IEC International Electrotechnical Commission. IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems.

[IGS02] S. Irani, R. Gupta, and S. Shukla. Competitive analysis of dynamic power management strategies for systems with multiple power savings states. In *Proceedings of the conference on Design, automation and test in Europe*, pages 117–, 2002.

[IP05] S. Irani and K. R. Pruhs. Algorithmic problems in power management. *SIGACT News*, 36 :63–76, June 2005.

[ISG03a] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 37–46, 2003.

[ISG03b] S. Irani, S. Shukla, and R. Gupta. Online strategies for dynamic power management in systems with multiple power-saving states. *ACM Transactions on Embedded Computing Systems*, 2 :325–346, August 2003.

[ITR] ITRS. International technology roadmap for semiconductors, 2010 update, overview, 2010.

[JG05] R. Jejurikar and R. Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings of the 42nd annual Design Automation Conference*, pages 111–116, 2005.

[JPG04] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41st annual Design Automation Conference*, pages 275–280, 2004.

[KAB+03] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current : Moore's law meets static power. *Computer*, 36(12) :68–75, December 2003.

[KL00] C. M. Krishna and Y.-H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium*, pages 156–, 2000.

[KWDY10] F. Kong, Y. Wang, Q. Deng, and W. Yi. Minimizing multi-resource energy for real-time systems with discrete operation modes. In *Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, pages 113–122, 2010.

[LB12] H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems*, pages 166–175, 2012.

[LDAVN08] M. Lemerre, V. David, C. Aussaguès, and G. Vidal-Naquet. Equivalence between schedule representations : Theory and applications. In *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 237–247, 2008.

[LJP13a] V. Legout, M. Jan, and L. Pautet. Mixed-criticality multiprocessor real-time systems : Energy consumption vs deadline misses. In *First Workshop on Real-Time Mixed Criticality Systems*, 2013.

[LJP13b] V. Legout, M. Jan, and L. Pautet. An off-line multiprocessor real-time scheduling algorithm to reduce static energy consumption. In *First Workshop on Highly-Reliable Power-Efficient Embedded Designs*, 2013.

[LJP13c] V. Legout, M. Jan, and L. Pautet. Réduction de la consommation statique des systèmes temps-réel multiprocesseurs. In *École d'Été Temps Réel 2013 (ETR'13)*, 2013.

[LJP13d] V. Legout, M. Jan, and L. Pautet. A scheduling algorithm to reduce the static energy consumption of multiprocessor real-time systems. In *21st International Conference on Real-Time Networks and Systems*, 2013.

[LL73] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20, January 1973.

[LRK03] Y.-H. Lee, K. Reddy, and C. Krishna. Scheduling techniques for reducing leakage power in hard real-time systems. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 105 – 112, 2003.

[LSH10] E. Le Sueur and G. Heiser. Dynamic voltage and frequency scaling : The laws of diminishing returns. In *Proceedings of the Workshop on Power Aware Computing and Systems*, pages 1–8, 2010.

[LW82] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4) :237 – 250, 1982.

[MAAM02] R. Melhem, N. AbouGhazaleh, H. Aydin, and D. Mossé. *Power management points in power-aware real-time systems*. 2002.

[MACM00] D. Mosse, H. Aydin, B. Childers, and R. Melhem. Compiler-assisted dynamic power-aware scheduling for real-time applications. In *In Workshop on Compilers and Operating Systems for Low Power*, 2000.

[MEA+10] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proceedings of the 2010 10th IEEE International Conference on Computer and Information Technology*, pages 1864–1871, 2010.

[MPC] *FreeScale MPC5510 Microcontroller Family Reference Manual*.

[MSD10] T. Mégel, R. Sirdey, and V. David. Minimizing task preemptions and migrations in multiprocessor optimal real-time schedules. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, pages 37–46, 2010.

[Mé12] T. Mégel. *Placement, ordonnancement et mécanismes de migration de tâches temps-réel pour des architectures distribuées multicœurs*. PhD thesis, Institut National Polytechnique de Toulouse – Université de Toulouse, 2012.

[NBGM11] G. Nelissen, V. Berten, J. Goossens, and D. Milojevic. Reducing preemptions and migrations in real-time multiprocessor scheduling algorithms by releasing the fairness. In *Proceedings of the 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 15 –24, 2011.

[NBN+12] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic. U-edf : An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 13 –23, 2012.

[NG06] N. Navet and B. Gaujal. Ordonnancement temps réel et minimisation de la consommation d'énergie. In *Systèmes temps réel 2 - Ordonnancement, réseaux et qualité de service*. 2006.

[NG09] V. Nélis and J. Goossens. Mora : an energy-aware slack reclamation scheme for scheduling sporadic real-time tasks upon multiprocessor platforms. In *Real-Time Computing Systems and Applications*, 2009.

[NGD+08] V. Nélis, J. Goossens, R. Devillers, D. Milojevic, and N. Navet. Power-aware real-time scheduling upon identical multiprocessor platforms. In *Proceedings of the 2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing*, pages 209–216, 2008.

[NQ04] L. Niu and G. Quan. Reducing both dynamic and leakage energy consumption for hard real-time systems. In *Proceedings of the 2004 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 140–148, 2004.

[Né10] V. Nélis. *Energy-Aware Real-Time Scheduling in Embedded Multiprocessor Systems*. PhD thesis, Université Libre de Bruxelles, 2010.

[PS01] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[QZ08] X. Qi and D. Zhu. Power management for real-time embedded systems on block-partitioned multicore platforms. In *Proceedings of the 2008 International Conference on Embedded Software and Systems*, pages 110–117, 2008.

[RLM+11] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt. Run : Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In *Proceedings of the IEEE 32nd Real-Time Systems Symposium*, pages 104–115, 2011.

[SBA+01] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli. Dynamic voltage scaling and power management for portable systems. In *Proceedings of the 38th annual Design Automation Conference*, pages 524–529, 2001.

[SBDM99] T. Simunic, L. Benini, and G. De Micheli. Cycle-accurate simulation of energy consumption in embedded systems. In *Proceedings of the 36th Design Automation Conference*, pages 867–872, 1999.

[SC05] V. Swaminathan and K. Chakrabarty. Pruning-based, energy-optimal, deterministic i/o device scheduling for hard real-time systems. *ACM Transactions on Embedded Computing Systems*, 4 :141–167, February 2005.

[SCS00] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 365–368, 2000.

[SJPL08] E. Seo, J. Jeong, S. Park, and J. Lee. Energy efficient scheduling of real-time tasks on multicore processors. *IEEE Transactions on Parallel and Distributed Systems*, 19(11) :1540–1552, 2008.

[SPH05] D. C. Snowdon, S. M. Petters, and G. Heiser. Power measurement as the basis for power management. In *Proceedings of the 1st Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2005.

[SRH05] D. Snowdon, S. Ruocco, and G. Heiser. Power management and dynamic voltage scaling : Myths and facts. In *2nd Int'l Workshop on Power-Aware Real-Time Computing*, 2005.

[Sta88] J. A. Stankovic. Misconceptions about real-time computing : A serious problem for next-generation systems. *Computer*, 21(10) :10–19, October 1988.

[STM] *ST Microelectronics STM32L151xx and STM32L152xx advanced ARM-based 32-bit MCUs Reference Manual*.

[Ves07] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, pages 239–243, 2007.

[WCL+07] H.-W. Wei, Y.-H. Chao, S.-S. Lin, K.-J. Lin, and W.-K. Shih. Current results on edzl scheduling for multiprocessor real-time systems. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 120–130, 2007.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3) :36 :1–36 :53, May 2008.

[WWDS94] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994.

[YDS95] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 374–, 1995.

[ZM07] Y. Zhu and F. Mueller. DVSleak : combining leakage reduction and voltage scaling in feedback edf scheduling. In *Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 31–40, 2007.

[ZMM03] D. Zhu, D. Mossé, and R. Melhem. Multiple-resource periodic scheduling problem : how much fairness is necessary ? In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, pages 142–, 2003.