Contributions à l'ordonnancement et l'analyse des systèmes temps réel critiques

François Dorin

THESE

pour l'obtention du Grade de

DOCTEUR DE L'ECOLE NATIONALE SUPERIEURE DE MECANIQUE ET D'AEROTECHNIQUE
(Diplôme National – Arrêté du 7 août 2006)

Ecole Doctorale : Sciences et Ingénierie pour l'Information
Secteur de Recherche : Informatique et Applications

Présenté par :

François DORIN

****

Contributions à l'ordonnancement et l'analyse des systèmes temps réel critiques

****

Directeurs de Thèse : Pascal RICHARD, Michaël RICHARD et Emmanuel GROLLEAU

****

Soutenue le 30 septembre 2010
devant la Commission d'Examen

****

JURY

Yvon TRINQUET	Professeur, IUT, IRCCyN	Président
Ye-Qiong SONG	Professeur, INPL/ENSEM, LORIA	Rapporteur
Patrick MARTINEAU	Professeur, Polytech'Tours, LI	Rapporteur
Pascal RICHARD	Professeur, ENSMA, LISI	Examinateur
Michaël RICHARD	Maître de Conférence, ENSMA, LISI	Examinateur
Emmanuel GROLLEAU	Maître de Conférence, ENSMA, LISI	Examinateur

Remerciements

Je tiens à remercier le Professeur Guy PIERRA, ancien directeur du LISI, ainsi que le Professeur Yamine AIT AMEUR, actuel directeur du LISI, pour m'avoir accueilli aussi chaleureusement et avoir mis à ma disposition tous les moyens techniques et scientifiques nécessaires à l'exercice de mon travail.

Je remercie également Pascal RICHARD, Michaël RICHARD et Emmanuel GROLLEAU, mes directeurs de thèse, sans qui tous mes travaux n'auraient pas pu être réalisés. Je les remercie de leur soutient, de leurs conseils lors de nos conversations, ainsi que de leurs qualités humaines et de leur écoute.

Merci aux Professeurs Ye-Qiong SONG et Patrick MARTINEAU d'avoir accepté la lourde tâche de rapporteur, ainsi qu'au Professeur Yvon TRINQUET pour l'honneur qu'il m'a fait en acceptant d'être le Président de mon jury.

Je voudrais également remercier mes parents et mon frère du soutien inconditionnel qu'ils m'ont apporté, ainsi qu'à Martine, que j'ai toujours considérée comme une deuxième mère. Je n'oublie bien évidement pas mes amis Stéphanie, Chedlia, Nabil, Aurélie et Cyril, qui ont été, bien malgré eux, des rochers sur lesquels je pouvais m'agripper dans les moments difficiles.

Il y a encore tant de personnes que je souhaiterais remercier : Christian, Valéry, Idir, Dago, Guillaume, Frédéric, Claudine, Youness, Yacine, Aurélie, Chimène, Sybille, Loé, Stéphane, Sadouanouan, Patrick, et tous les membres du LISI en général. A toutes ces personnes, je souhaiterais leur dire merci pour les merveilleux moments que j'ai passés en leur compagnie.

Table des matières

Partie I Etat de l'art

Table des figures

Liste des tableaux

Introduction générale

L'utilisation de l'informatique temps réel ne cesse de se répandre dans notre quotidien, que ce soit au travers de la téléphonie mobile, de systèmes de contrôle embarqués, l'électroménager ou le multimédia. L'idée sous-jacente des systèmes temps réel n'est pas la notion de rapidité mais de réactivité : un système temps réel évolue dans un environnement dynamique, et doit donc s'adapter en permanence aux changements survenant dans cet environnement. Ceci induit donc que la réponse à ces changements doit être adaptée (correction fonctionnelle), mais doit également respecter des contraintes de temps (correction temporelle), de part la nature dynamique de l'environnement dans lequel évolue le système.
Les systèmes temps réel s'immisçant de plus en plus dans notre quotidien gagne en complexité, que ce soit en complexité architecturale (nombre de processeurs, présence de réseaux), ou en complexité des exigences (criticité des tâches, consommation électrique, etc...). C'est dans ce contexte de complexité grandissante que se situent les travaux présentés dans le cadre de cette thèse.
Dans la partie état de l'art, nous introduirons les systèmes temps réel, ainsi que la problématique de l'ordonnancement. Nous présenterons des résultats de base qui ont été développés dans la littérature dans le but de répondre à cette problématique. Nous introduirons également la notion de validation, réalisée durant la conception d'un système temps réel afin de prouver la correction temporelle des applications. Nous présenterons les méthodes de validation qui ont été utilisées dans le cadre de nos contributions, avec notamment le calcul des pires temps de réponse en priorité fixe monoprocesseur, l'analyse de la demande processeur et l'analyse holistique, prenant en compte les dépendances entre les tâches et les messages, dans le cas des architectures distribuées.
Dans le chapitre 3, nous présenterons une solution pour simultanément placer et ordonnancer des tâches dans un système temps réel distribué, tout en minimisant le nombre de processeurs nécessaires pour respecter les spécifications temporelles des tâches. Le principe de l'algorithme repose sur l'énumération implicite (méthode de type Branch and Bound) de tous les ordon-

nancements possibles. La technique de validation utilisée est un dérivé de l'analyse holistique, qui permet d'obtenir une borne inférieure des pires temps de réponse des tâches et des messages durant le processus de placement et d'affectation des priorités, et ainsi d'éliminer les ordonnancements non valides.
Dans le chapitre 4, nous étudierons le modèle de tâches à criticité multiple. L'objectif de ce modèle est de prendre en compte la notion de criticité, telle qu'elle a été introduite par exemple dans la norme DO-178B utilisée en aéronautique. En effet, ce standard définit plusieurs niveaux de criticité en fonction des conséquences qu'aurait une défaillance temporelle sur le système, en allant de la tâche critique, dont une défaillance provoquerait le crash d'un avion, à la tâche non critique, dont la défaillance ne serait que peu ou pas notable. L'objectif est donc de pouvoir prendre en compte cette notion de criticité durant la validation du système. Nous avons prouvé que dans ce cadre, d'une part que l'algorithme d'Audsley est optimal et d'autre part que l'algorithme développé par Vestal minimise la vitesse du processeur nécessaire pour ordonnancer fiablement les tâches. Nous avons également adapté la méthode d'analyse de sensibilité développée par Bini pour déterminer les variations admissibles des durées d'exécution des tâches.
Dans le chapitre 5, nous nous intéresserons aux systèmes multiprocesseurs ordonnancés par des algorithmes d'ordonnancement semi-partitionné. Cette classe d'algorithme utilise une stratégie par partitionnement pour répartir les tâches parmi les processeurs tout en autorisant la migration des tâches qui ne peuvent pas être affectées à un processeur donné sans violation d'échéance. Nous avons développé des algorithmes semi-partitionnés en priorité dynamique (EDF), autorisant la migration des tâches, mais interdisant la migration des instances, afin de limiter le surcoût processeur induit par les migrations. Le principe de nos algorithmes est de dupliquer chaque tâche migrante sur chacun des processeurs devant exécuter au moins une instance de cette tâche, en utilisant une tâche multiframe. Nous avons comparé nos algorithmes avec l'algorithme First Fit Decreasing (systèmes partitionnés) et avec un autre algorithme de semi-partitionnement, développé par Shinpei Kato (qui autorise les migrations des instances de tâches en cours d'exécution).

Première partie

Etat de l'art

Chapitre

1 Introduction

Sommaire

Résumé

Ce chapitre est une introduction aux systèmes temps réels. Nous y introduirons l'ensemble des notions nécessaires à la compréhension des chapitres suivants. Ce chapitre est composé d'une première partie décrivant les systèmes temps réel puis d'un état de l'art sur les notions qui seront utilisées : architectures monoprocesseurs, multiprocesseurs et distribuées, partage de ressources, contraintes de précédence, etc.

I. Informatique temps réel

L'informatique occupe une place de plus en plus importante dans notre société, avec l'expansion de systèmes informatisés : on peut penser aux ordinateurs, bien entendu, mais également à des objets que nous utilisons tous les jours : téléphones portables, voitures, appareils électroménagers, etc...

Tous ces systèmes se doivent d'être *fonctionnellement correct*, c'est-à-dire faire ce pour quoi ils sont faits. Par exemple, une calculatrice se doit de faire des calculs justes.

Ce qui distingue l'informatique temps-réel de l'informatique "classique", c'est la prise en compte de la notion de *temps*. Pour bien comprendre comment ce temps est pris en compte, arrêtons-nous sur ces deux mots, *temps* et *réel*, comme l'a fait Buttazzo dans [But97] :

- Le nom *temps* est là pour signifier que l'exactitude d'un système dépend non seulement du résultat d'un calcul, mais également de la date à laquelle le résultat est produit.
- L'adjectif *réel* est là pour indiquer que la réaction d'un système à des événements externes doit survenir pendant leurs évolutions.

Pour bien comprendre cette nécessité, prenons l'exemple de l'ABS [1]. Ce système doit se déclencher dès que les roues de la voiture se bloquent lors d'un freinage brusque, et non plusieurs secondes après. Ainsi donc, un tel système se doit d'être correct fonctionnellement (i.e., se déclencher en cas de blocage des roues) et temporellement (i.e., se déclencher dès que le blocage des roues a lieu).

Ce type de système est présent dans de nombreux domaines. Sans être exhaustif, nous pouvons citer entre autres le contrôle de procédés industriels (centrale nucléaire [NSKT98], sidérurgie [Wal97], etc...), les applications multimédias [Owe97] ou encore les systèmes embarqués (aéronautique et aérospatiale [Bla09], automobile [AUT, FMB+09], etc...).

Une erreur courante, due à l'utilisation fréquente de l'expression *temps réel* de nos jours, est de croire qu'un système temps réel doit agir vite. En réalité, la notion de vitesse dépend de l'environnement dans lequel évolue le système.

Par exemple, un pilote automatique permettra à un petit robot autonome de réaliser un parcours tout en évitant des obstacles statiques (comme un mur) ou lents (comme une personne marchant). Mais si un événement "rapide" survient, c'est-à-dire qui évolue plus rapidement que ce que le robot peut prendre en compte, comme la présence d'une personne en train de courir au lieu de marcher, alors le robot peut ne pas avoir le temps de réagir et donc ne pas avoir le temps d'éviter la collision.

Cet exemple montre que le concept de temps n'est pas une propriété du système de contrôle (le robot), mais est lié à l'environnement dans lequel évolue le système. Et si un événement survient plus vite que ce que le système de contrôle peut prendre en compte, alors cela met en danger la survie du système.

Nous venons de voir un exemple de système temps réel. Une formalisation de cette notion est donnée figure 1.1. Le système de contrôle agît sur son environnement à travers des actionneurs,

1. Anti-Blocking System

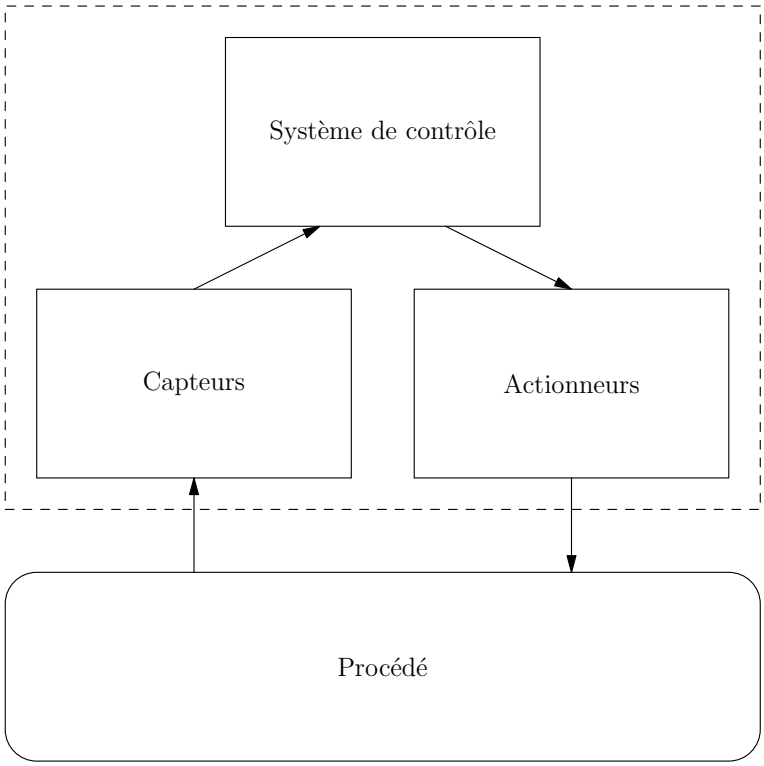


FIGURE 1.1 – Système temps réel

et récupère des informations sur l'état de l'environnement grâce à des capteurs.

Il existe une classification répandue des systèmes temps réel en fonction des conséquences que peut avoir le non respect d'une échéance :

- un système temps réel sera *à contraintes strictes (hard-real-time system)*, si le non respect d'une échéance a des conséquences catastrophiques sur le système temps réel, au point de pouvoir nuire à son intégrité.
- un système temps réel sera *à contraintes non strictes (soft-real-time system)*, si le non respect d'une échéance produit seulement une altération de la qualité du résultat, sans pour autant nuire à l'intégrité du système (par exemple, application multimédia).

II. Architecture des systèmes temps réel

Un système temps réel est constitué d'une *couche logicielle* s'exécutant sur un ou des calculateur(s) formant la *couche matérielle*.

II.1. Architecture matérielle

Le terme *architecture matérielle* désigne ici l'ensemble des ressources matérielles (ou physiques) qui peuvent être nécessaire à l'exécution de la couche logicielle : cela inclut donc les processeurs, mais également la mémoire, les réseaux, les cartes d'entrées / sorties (qui sont reliées, par exemple, aux capteurs et actionneurs), support de stockage, etc.
Les architectures peuvent être classées dans trois grandes catégories en fonction de leur composition, et notamment en fonction du nombre de processeurs utilisés et à la présence ou non de réseaux :

- *architecture monoprocesseur* : la couche matérielle est dans ce cas composée d'un unique processeur et il n'y a pas de réseaux. Ceci implique que le processeur exécute toutes les applications composant le système temps réel, le temps processeur étant alors partagé entre les différentes applications ;
- *architecture multiprocesseur* : la couche matérielle est composée de plusieurs processeurs partageant une même mémoire centrale. Les applications du systèmes sont donc réparties sur ces différents processeurs. Notons, comme pour l'architecture monoprocesseur, l'absence de réseaux.
- *architecture distribuée* : la couche matérielle est composée de plusieurs processeurs, chaque processeur ayant une mémoire qui lui est propre. Il n'y a donc pas de mémoire commune comme dans le cas des architectures multiprocesseurs. Des applications se trouvant sur des processeurs différents et ayant besoin de communiquer entre elles le feront à travers un réseau.

II.1.a. Les processeurs

Un processeur est une unité de calcul. Sauf avis contraire, lorsque nous parlerons d'architectures multiprocesseurs ou distribuées, tous les processeurs seront identiques, c'est-à-dire qu'ils seront interchangeables : une application pourra s'exécuter aussi bien sur un processeur que sur un autre. De tels systèmes sont dit *homogènes*.
Dans le cas où un système est composé de processeurs dont au moins une caractéristique diffère d'un processeur à l'autre (vitesse de calcul, mémoire, jeu d'instruction, etc...), le système est dit *hétérogène*.

II.1.b. Les réseaux

Les réseaux sont le moyen de communication utilisé par les tâches. Deux tâches se trouvant sur des processeurs différents et souhaitant communiquer le font en envoyant des données sous forme de *messages* qui transitent par les réseaux.
Ils existent plusieurs types de réseaux, dont voici une liste non exhaustive :

- le réseau CAN [ISO94a] ;
- le réseau VAN [ISO94b] ;
- le réseau AFDX / ARINC [Ari09].

Certains de ces réseaux sont très employés dans un domaine particulier. Par exemple, le type de réseau CAN est largement employé dans l'industrie automobile, tandis que le type de réseau AFDX est plutôt employé dans l'industrie aéronautique et spatiale.

II.2. Architecture logicielle

L'architecture logicielle d'un système temps réel se décompose typiquement en deux parties distinctes :
- un *exécutif temps réel*, qui est une partie de bas niveau faisant le lien avec la couche matérielle ;
- un *programme applicatif*, qui est une partie de haut niveau, correspondant aux fonctions permettant de contrôler le système temps réel.

II.2.a. Exécutif temps réel

L'exécutif temps réel est composé de deux parties :
- un *noyau temps réel*, qui est le coeur de l'exécutif temps réel ;
- de *modules* ou *bibliothèques* venant compléter le noyau temps réel et facilitant la conception d'une application à travers l'apport de routines de gestion de fichiers, de gestion de timers, de gestion de réseaux, etc...

Un noyau, pour pouvoir être qualifié de temps réel, doit répondre à certaines exigences, notamment en fournissant des services de base [But97] :
- ordonnancement ;
- routines de gestion de ressources ;
- primitives de communication.

En fonction de la politique d'ordonnancement retenue, on parlera d'*ordonnanceur* (cas d'une politique en ligne), ou de *séquenceur* (cas d'une politique hors ligne). Les notions de politiques *en-ligne* et *hors-ligne* seront définies ultérieurement dans le paragraphe II.2.j.

En plus de ces services, un exécutif temps réel se doit de fournir des garanties, notamment concernant le temps d'exécution nécessaire aux différentes primitives fournies par l'exécutif temps réel (ou plus précisément, des bornes supérieures de temps d'exécution), afin de permettre d'estimer le temps d'exécution nécessaire pour chaque tâche du système temps réel.

Voici une liste de quelques exécutifs temps réel utilisés dans les systèmes embarqués :
- VxWorks [Riv]
- RT Linux [Bar97]
- Ada Temps Réel [Ros, BDV04]
- Osek / VDK [Ose05]
- POSIX [IEE04]

Concernant les derniers exécutifs temps réel (Ada, Osek / VDX et Posix), il s'agit en réalité de normes définissant des spécifications et qu'une implémentation se doit de suivre afin de respecter lesdites normes.

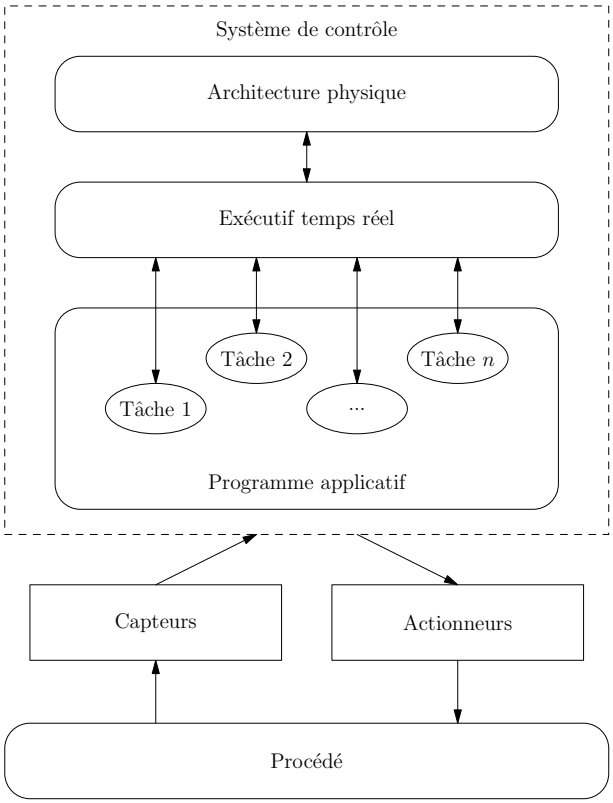


FIGURE 1.2 – Structure de l'architecture logicielle d'un système temps réel

II.2.b. Programme applicatif

Le programme applicatif correspond à la partie logicielle du système qui va exécuter les différentes fonctions nécessaires au contrôle du système, et plus particulièrement du procédé. Le programme applicatif est divisé en entités distinctes appelées *tâches*, chacune ayant un rôle qui lui est propre, comme par exemple :
– réaliser un calcul ;
– être associé à une alarme ;
– traiter des entrées / sorties ;
– etc...
La notion de *tâche* que nous venons d'introduire joue un rôle clé dans les systèmes temps réel, puisque ce sont ces entités de base qui composent le programme applicatif d'un système temps réel. L'exécutif temps réel met à disposition des routines que peuvent utiliser les tâches pour réaliser leur fonction. Ces routines permettent notamment :
– de gérer les communications entre tâches ;
– de réaliser de l'acquisition de données en provenance des capteurs ;
– d'actionner des commandes sur le système contrôlé, à l'aide des actionneurs.

Nous avons vu que la caractéristique majeure d'un système temps réel était la prise en compte du temps. Le temps est pris en compte grâce à la définition de caractéristiques et de contraintes temporelles au niveau des tâches. Il est possible de classer les tâches en différentes catégories, en fonction de leurs caractéristiques :
- une tâche *périodique* est une tâche dont l'activation est régulière et le délai entre deux activations successives est constant. Ce type de tâche est généralement utilisé pour des tâches de contrôle de procédé.
- une tâche *sporadique* est une tâche caractérisée par un délai minimum entre deux activations successives. Ainsi, la pire charge induite par une tâche sporadique est connue a priori.
- une tâche *apériodique* est une tâche dont on ne connaît aucune caractéristique. Elle est généralement activée sur un événement extérieur.

Cette classification nous amène également à introduire la notion d'*instance*, et de bien distinguer cette notion avec celle de tâche :
- une *tâche* correspond à du code exécutable, sur lequel sont définies des contraintes ;
- une *instance* d'une tâche correspond à une occurrence (ou exécution) de cette tâche.

Cette prise en compte induit l'utilisation sous-jacente de modèles. Dans le paragraphe qui suit, nous allons donc logiquement introduire un modèle de tâches que nous utiliserons dans le cadre de cette thèse : le *modèle de tâche périodique*.

II.2.c. Modèle de tâche périodique

Le modèle de tâche périodique présenté est celui défini par Liu et Layland dans [LL73], qui définit une tâche τ_i comme étant un triplet (C_i, D_i, T_i), où :
- C_i est la *pire durée d'exécution* de la tâche τ_i. Cela correspond au temps processeur maximum nécessaire à l'exécution d'une instance de la tâche τ_i. La détermination du temps d'exécution d'une tâche est un problème à part entière ayant été et continuant d'être étudié dans la littérature [WEE+08].
- D_i est l'*échéance relative* de la tâche τ_i. Une instance j de la tâche τ_i activée à l'instant t doit avoir terminée son exécution avant l'instant $t + D_i$. Cet instant $t + D_i$ est également appelé *échéance absolue* de l'instance j de τ_i et est notée $d_{i,j}$.
- T_i est la *période* de la tâche τ_i. Si une instance de la tâche τ_i est activée à l'instant t, alors la prochaine instance de la tâche τ_i sera activée à l'instant $t + T_i$.

Chaque caractéristique est représentée visuellement sur la figure 1.3.
En fonction de la valeur de l'échéance relative D_i par rapport à la période T_i de chaque tâche d'un système, on dit qu'un système de tâches τ est un système de tâches :
- *à échéance sur requête*, si l'échéance relative est égale à la période pour chaque tâche, i.e., $D_i = T_i, \forall \tau_i \in \tau$;
- *à échéance contrainte*, si l'échéance relative est inférieure ou égale à la période pour chaque tâche, i.e., $D_i \leq T_i, \forall \tau_i \in \tau$;
- *à échéance arbitraire*, si l'échéance relative n'est pas reliée à la période, c'est-à-dire qu'il existe une tâche τ_i du système ayant une échéance D_i supérieure à sa période T_i.

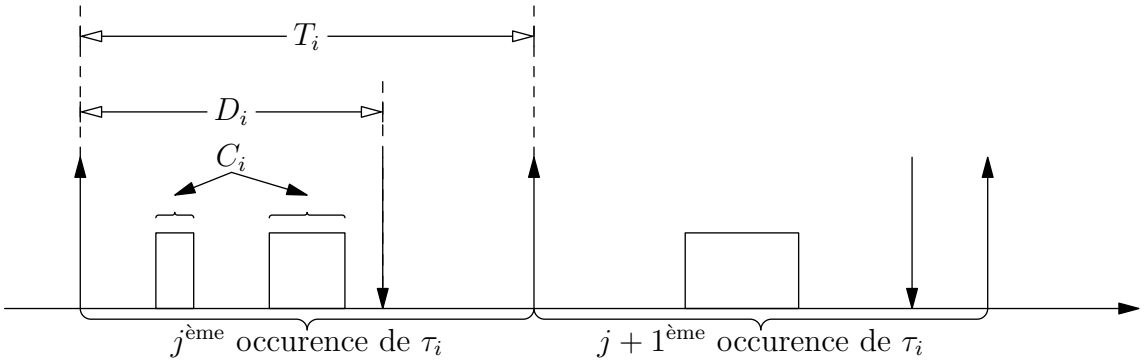


FIGURE 1.3 – Représentation graphique du modèle de Liu et Layland

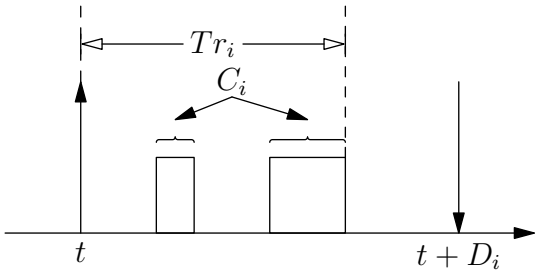


FIGURE 1.4 – Temps de réponse d'une instance d'une tâche τ_i

II.2.d. Système synchrone / asynchrone

Un système de tâche sera dit *synchrone* si toutes les tâches le composant démarrent leur exécution au même instant. Sans perte de généralité, cet instant est habituellement pris comme instant de référence, c'est-à-dire l'instant 0.
Dans le cas contraire, c'est-à-dire lorsqu'il existe au moins une tâche démarrant de manière différée par rapport aux autres tâches, alors le système de tâches sera dit *asynchrone*.

II.2.e. Temps de réponse

La notion de *temps de réponse* est une notion souvent utilisée pour vérifier l'ordonnançabilité d'un système de tâches temps réel. Nous allons donc en donner une définition ci-dessous.

Définition 1. *Le temps de réponse d'une instance d'une tâche τ_i est défini comme étant le délai entre la date d'activation de l'instance et sa date de fin d'exécution (cf. figure 1.4).*

Cette définition nous permet de définir le pire temps de réponse d'une tâche :

Définition 2. *Le pire temps de réponse d'une tâche τ_i, noté Tr_i, est le plus grand temps de réponse parmi toutes les instances de τ_i.*

Autrement dit, le temps de réponse d'une instance de la tâche τ_i est forcément inférieur ou égal au pire temps de réponse de la tâche τ_i.

II.2.f. Ordonnançabilité

Lorsqu'un système temps réel est exécuté sur une architecture donnée, nous devons avoir un moyen fiable de décider, à un instant donné, quelle tâche va s'exécuter, de sorte que toutes les contraintes temporelles soient vérifiées : c'est l'objectif de l'*ordonnancement*. Autrement dit, l'ordonnancement est la technique qui consiste à organiser l'exécution des instances des tâches de sorte que toutes les instances terminent leur exécution avant d'atteindre leur échéance, et ainsi garantir l'intégrité du système. Ce qui nous permet d'introduire les définitions suivantes :

Définition 3. *Une tâche τ_i est dite ordonnançable ou valide lorsque toutes les instances de la tâche τ_i respectent leur échéance.*

Et nous pouvons maintenant étendre cette définition au système temps réel tout entier :

Définition 4. *Un système temps réel est dit ordonnançable ou valide lorsque toutes les tâches qui le composent sont elles-mêmes ordonnançables.*

Lorsqu'on parle d'ordonnancement, il est également possible d'inclure d'autres critères, comme des critères d'optimisation des temps de réponses ou de réduction de consommation énergétique.

II.2.g. Optimalité

Une notion importante, lorsqu'on étudie des algorithmes d'ordonnancement, est la notion d'optimalité.

Définition 5. *Un algorithme $\mathcal{A}$ est dit optimal si, et seulement si, il peut ordonnancer fiablement tout système de tâches ordonnançables.*

C'est-à-dire que si un système de tâches τ est ordonnançable par un algorithme quelconque $\mathcal{A}'$, alors l'algorithme $\mathcal{A}$ pourra ordonnancer fiablement τ. Il est à noter qu'un algorithme ne sera dit optimal que dans un cadre précis, comme par exemple, pour un système de tâches indépendantes, synchrones et à échéance sur requête.

II.2.h. Dominance / non comparabilité

Il peut être utile parfois de pouvoir comparer des algorithmes entre eux. Cela peut être fait à travers deux notions, celle de dominance, et celle de non-comparabilité.

Définition 6. *Un algorithme $\mathcal{A}$ domine un algorithme $\mathcal{B}$ si tout système de tâches ordonnançable par $\mathcal{B}$ l'est par $\mathcal{A}$ et s'il existe au moins un système de tâche ordonnançable par $\mathcal{A}$ et qui ne le soit pas par $\mathcal{B}$.*

Définition 7. *Deux algorithmes $\mathcal{A}$ et $\mathcal{B}$ seront dit non comparable si, et seulement si :*
– *il existe un système de tâches τ qui soit ordonnançable par $\mathcal{A}$ et non ordonnançable par $\mathcal{B}$;*
– *il existe un système de tâches τ' qui soit ordonnançable par $\mathcal{B}$ et non ordonnançable par $\mathcal{A}$;*

II.2.i. Notion de facteur d'utilisation

Une notion très utilisée est la notion de facteur d'utilisation, car de nombreuses conditions nécessaires et / ou suffisantes d'ordonnançabilité l'utilisent :

Définition 8. *Le facteur d'utilisation d'une tâche τ_i est définie comme étant le rapport entre son temps d'exécution C_i et sa période T_i. Il est noté u_i :*

$$u_i = \frac{C_i}{T_i} \tag{1.1}$$

Le facteur d'utilisation d'une tâche correspond au temps processeur nécessaire pour l'exécution correct de la tâche. Par exemple, un facteur d'utilisation de 0.2 indique que la tâche requiert 20% du temps processeur pour s'exécuter dans les temps.
Par extension, on peut définir le facteur d'utilisation d'un système de tâches :

Théorème 1. *Le facteur d'utilisation d'un système de tâches correspond à la somme des facteurs d'utilisation des tâches composant le système :*

$$U = \sum_{i=1}^{n} u_i \tag{1.2}$$

De la définition du facteur d'utilisation découle une condition nécessaire à l'ordonnancement d'un système sur un processeur :

Théorème 2. *Un système de tâche τ ne peut être ordonnançable sur m processeurs si son facteur d'utilisation est strictement supérieure à m.*

En effet, dans ce cas, on demande aux processeurs plus de puissance qu'ils ne peuvent en fournir. Le système est donc forcément non ordonnançable.

II.2.j. Algorithme en-ligne / hors-ligne

Les algorithmes d'ordonnancement peuvent être classés dans deux catégories, en fonction de leur nature :
– algorithme d'*ordonnancement hors-ligne* : l'ordonnancement consiste à définir une séquence d'exécution qui sera ensuite répétée indéfiniment par le séquenceur. La séquence est issue d'un calcul effectué hors-ligne, c'est-à-dire *a priori* de l'exécution du système de tâches.
– algorithme d'*ordonnancement en-ligne* : l'ordonnancement consiste à déterminer la séquence d'exécution durant la vie du système temps-réel. L'ordonnanceur choisit quelle tâche doit s'exécuter en fonction des caractéristiques des tâches.
Les travaux de cette thèse se placent dans le cadre des *ordonnancements en-ligne.*

II.2.k. Priorité fixe / dynamique

La majorité des algorithmes en ligne s'appuie sur une politique d'affectation de priorité pour l'ordonnancement des tâches : le processeur est donné à la tâche ayant la plus grande priorité. La convention retenue dans le cadre de cette thèse est de représenter la priorité de la tâche τ_i par un entier π_i, et de considérer que plus π_i est grand, plus faible est la priorité de τ_i. Avec cette convention, une valeur de 0 pour π_i indique donc que la tâche τ_i est la plus prioritaire.
L'étape d'ordonnancement consiste alors à affecter des priorités aux tâches, qui peuvent être :
- *à priorité fixe* : une tâche est affectée d'une priorité. Toutes les instances de cette tâche hériteront de cette priorité.
- *à priorité dynamique* : la priorité n'est plus affectée à une tâche mais aux instances. Ici encore, il est possible de distinguer deux types de priorité :
  - FJP (*Fixed Job Priority*) : la priorité d'une instance d'une tâche est constante ;
  - DP (*Dynamic Priority*) : la priorité d'une instance peut évoluer au cours du temps.

II.2.l. Caractère préemptif / non préemptif

Le caractère préemptif ou non préemptif des tâches modifie la façon dont elles sont agencées :
- une tâche *préemptible* peut, à tout moment, être interrompue pour permettre à une autre tâche de s'exécuter (par exemple, parce qu'elle est plus prioritaire).
- une tâche *non préemptible* ne peut être interrompue à partir du moment où elle a commencé son exécution, même si une tâche plus prioritaire demande à avoir le processeur. Dans ce cas, la tâche de plus forte priorité devra attendre que la tâche de plus faible priorité ait terminé son exécution pour pouvoir à son tour s'exécuter.

III. Ordonnancement monoprocesseur

Un ordonnanceur est un algorithme de l'exécutif temps réel qui attribue à chaque instant le processeur à la tâche la plus prioritaire.

III.1. Priorité fixe

Il existe de nombreux algorithmes d'attribution des priorités fixes, et les paragraphes suivant sont consacrés à la description de quelques-uns d'entre eux.

III.1.a. Rate Monotonic

Rate Monotonic est un algorithme à priorité fixe introduit par Liu et Layland dans [LL73]. Cet algorithme affecte des priorités aux tâches en fonction de leur période : plus leur période est

Tâche	P_i	C_i	π_i
τ_1	10	1	3
τ_2	4	1	0
τ_3	5	1	1
τ_4	8	2	2

TABLE 1.1 – Affectation des priorités selon Rate Monotonic

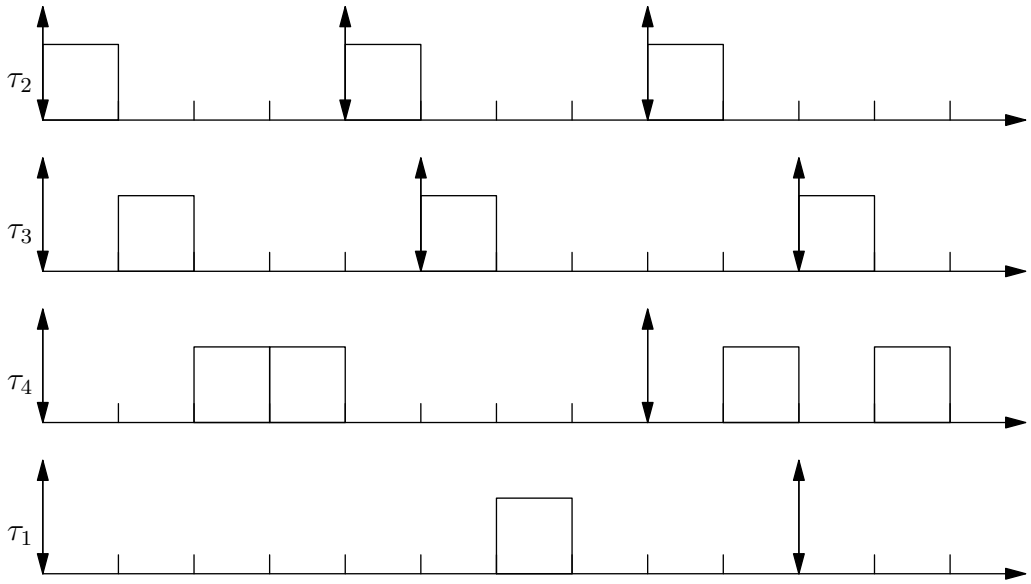


FIGURE 1.5 – Ordonnancement selon Rate Monotonic

petite, plus la tâche est prioritaire. Un exemple de système de tâche ordonnancée par Rate Monotonic est donné table 1.1. La figure 1.5 est une représentation graphique de l'ordonnancement correspondant.

Optimalité Cet algorithme est optimal dans le cadre résumé par le théorème 3 :

Théorème 3 ([LL73]). *Rate Monotonic est optimal pour l'ordonnancement de systèmes de tâches synchrones, indépendantes et à échéance sur requête en présence de préemption.*

Conditions suffisantes d'ordonnançabilité Liu et Layland ont donné dans [LL73], la condition suffisante d'ordonnançabilité :

Théorème 4 ([LL73]). *Un système temps réel composé de n tâches est ordonnançable par Rate Monotonic si la condition suivante est vérifiée :*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{1.3}$$

Une forme dérivée fréquemment utilisée pour sa simplicité, même si elle est un peu moins précise, est obtenue à partir de l'équation 1.3 en faisant tendre n vers l'infini :

$$U \leq \frac{1}{\ln 2} \approx 0.69 \tag{1.4}$$

Il est à noter que les conditions précédentes sont des conditions *suffisantes* d'ordonnançabilité, ce qui signifie que si ces conditions ne sont pas vérifiées, alors elles ne permettent pas de conclure quant à l'ordonnançabilité du système de tâches étudié (il peut être ordonnançable comme non ordonnançable).

III.1.b. Deadline Monotonic

Deadline Monotonic est un algorithme à priorité fixe introduit par Leung et Whitehead dans [LW82]. Cet algorithme est proche de celui de Rate Monotonic, à la différence que les priorités sont maintenant affectées en fonction de l'échéance relative de chaque tâche au lieu de leur période.

Optimalité Tout comme Rate Monotonic, cet algorithme est optimal dans un cadre explicité par le théorème 5 :

Théorème 5 ([LW82]). *Cet algorithme est optimal dans le cadre des algorithmes à priorité fixe pour des systèmes de tâches synchrones à échéance contrainte lorsque la préemption est autorisée.*

Notons également que lorsque les tâches d'un système sont à échéance sur requête, alors Rate Monotonic et Deadline Monotonic se confondent.

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

Théorème 6. *Un système temps réel composé de n tâches est ordonnançable par Deadline Monotonic si la condition suivante est vérifiée :*

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{1.5}$$

Tâche	P_i	D_i	C_i	π_i
τ_1	10	5	1	2
τ_2	4	3	1	0
τ_3	5	4	1	1
τ_4	8	7	2	3

TABLE 1.2 – Affectation des priorités selon Deadline Monotonic

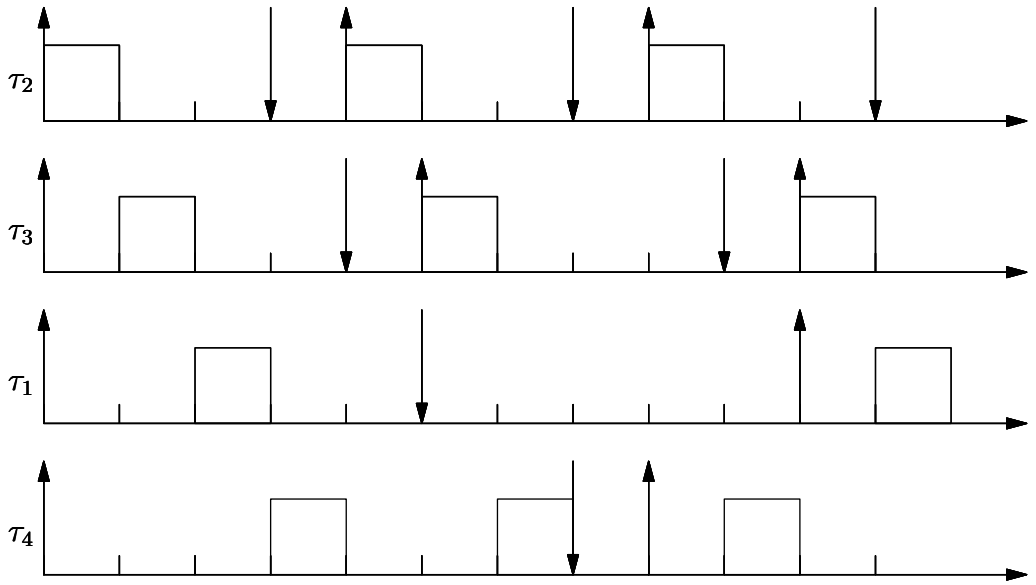


FIGURE 1.6 – Ordonnancement selon Deadline Monotonic

III.1.c. Audsley

Lorsque nous considérons un système de tâches à échéance arbitraire, les algorithmes précédents ne sont plus optimaux. Audsley, dans [Aud91], a introduit un nouvel algorithme qui a la particularité d'être optimal dans ce cadre, comme le résume le théorème 7 :

Théorème 7 ([Aud91]). *L'algorithme d'Audsley est optimal dans le cadre de systèmes de tâches asynchrones, indépendantes et à échéance arbitraire en présence de préemption.*

Cet algorithme est basé sur le constat suivant : l'interférence due aux tâches plus prioritaires dépend uniquement des tâches qui composent l'ensemble des tâches plus prioritaires, et non de leur priorité relative. L'idée d'Audsley est alors la suivante : il s'agit de parcourir tous les niveaux de priorité, du moins prioritaire au plus prioritaire. A chaque niveau de priorité est assignée la première tâche qui est ordonnançable à ce niveau. S'il y a au moins un niveau de priorité pour lequel aucune tâche ne peut être assignée, alors le système est non ordonnançable.

III.2. Priorité dynamique

Les algorithmes à priorité dynamique affectent une priorité qui n'est plus une donnée statique. La priorité d'une tâche est mise à jour durant la vie du système en fonction de certains critères, les critères utilisés dépendant de l'algorithme utilisé. Nous allons ainsi voir, dans les paragraphes suivant, deux algorithmes bien connus en priorité dynamique : *Earliest Deadline First* (EDF) et *Last Laxity* (LL).

III.2.a. Earliest Deadline First

Earliest Deadline First est un algorithme connu et étudié depuis longtemps [LL73, Der74, Hor74]. Le principe de cet algorithme est d'accorder la priorité la plus grande à la tâche ayant une instance dont l'échéance absolue est la plus proche.
L'avantage majeur de cet algorithme est qu'en présence d'un système de tâche à échéance sur requête, le taux d'utilisation maximum du processeur est de 100% (théorème 8).

Théorème 8 ([LL73]). *Un système de n tâches à échéance contrainte est ordonnançable par Earliest Deadline First si et seulement si :*

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{1.6}$$

Notons également que cet algorithme est optimal :

Théorème 9 ([Der74]). *Earliest Deadline First est optimal pour ordonnancer des systèmes de tâches indépendantes lorsque le facteur d'utilisation U du système est inférieure ou égale à 1 (absence de surcharge).*

III.2.b. Least Laxity

L'algorithme Least Laxity [Mok83] utilise la notion de *laxité* pour attribuer des priorités aux tâches.

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

Par exemple, une tâche avec une laxité de 0 doit obligatoirement avoir le processeur jusqu'à sa terminaison sans quoi elle ratera son échéance et sera donc non ordonnançable.
Le principe de l'algorithme est donc d'attribuer la plus haute priorité à l'instance dont la laxité est la plus faible, car c'est l'instance ayant le moins de marge possible. Il est à noter que d'une part, cet algorithme nécessite une mise à jour des priorités des tâches à chaque instant, et mobilise dans ce but beaucoup de ressources de calcul, mais qu'en plus, il provoque de nombreux changements de contexte, également coûteux en temps.

Théorème 10 ([Mok83]). *Least Laxity est optimal pour ordonnancer des systèmes de tâches indépendantes lorsque la charge du système U est inférieure ou égale à 1.*

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

III.3. Gestion des ressources

Jusqu'à présent, nous avons fait l'hypothèse que les systèmes de tâches traités étaient constitués de tâches indépendantes. Dans de nombreux applicatifs, les tâches peuvent partager des ressources logicielles (comme des accès mémoire) ou matérielles (comme des capteurs). La caractéristique d'une ressource concerne son utilisation réglementée. Par exemple, une ressource dite critique ne peut être utilisée que par une seule tâche à la fois. Si une tâche τ_i veut prendre une ressource et que celle-ci n'est pas disponible, alors la tâche τ_i est mise en attente, jusqu'à ce que la ressource soit à nouveau disponible.

La présence de ressources au sein d'un système temps réel peut être la cause :

- d'*inversions de priorité* : une inversion de priorité a lieu lorsqu'une tâche est retardée par une tâche de plus faible priorité alors qu'elles ne partagent pas de ressources. Sur la figure 1.7, la tâche τ_1 ne peut s'exécuter à l'instant $t = 2$ car la ressource R est possédée par τ_3. τ_1, ne pouvant pas avoir la ressource détenue par τ_3, est bloquée. τ_2, ayant une priorité plus forte que τ_3, s'exécute au dépend de τ_1 qui est en attente et qui a une priorité plus forte que celle de τ_2. On voit donc, sur cette exemple, que τ_2 retarde l'exécution de τ_1 alors que τ_1 et τ_2 ne partagent pas de ressources.
- d'*interblocages* : un interblocage peut se produire si une tâche τ_i veut prendre une ressource R_1 possédée par une tâche τ_j, et que τ_j veuille prendre une ressource R_2 possédée par τ_i. Les deux tâches sont bloquées, car τ_i, pour prendre R_1 doit attendre que la tâche τ_j la libère. Mais la tâche τ_j, pour pouvoir libérer la ressource R_1, doit continuer son exécution, ce qui lui est impossible tant que τ_i n'a pas libéré la ressource R_2. Les deux tâches τ_i et τ_j sont interbloquées.
- d'*anomalies d'ordonnancement* : pour des systèmes de tâches indépendantes, la réduction du temps d'exécution d'une tâche ne peut qu'améliorer les temps de réponses des tâches. Autrement dit, un système ordonnançable reste ordonnançable. En présence de ressources partagées, la réduction du temps d'exécution d'une tâche peut rendre un système ordonnançable non ordonnançable (cf. figure 1.8).

Afin de gérer ces différents effets, des protocoles de gestion des ressources ont été mis au point.

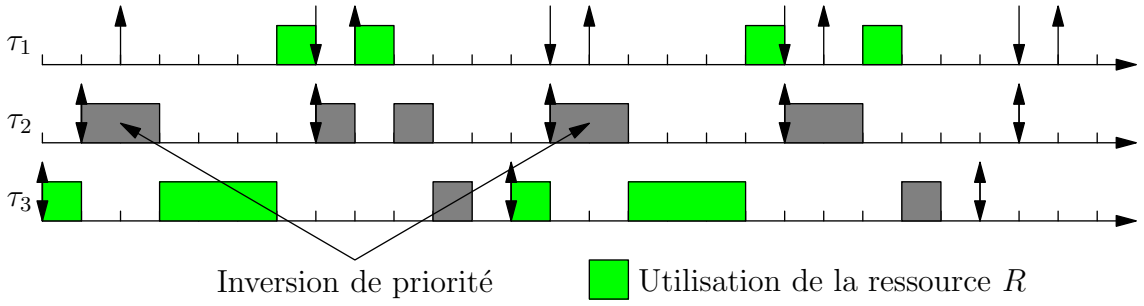


FIGURE 1.7 – Inversion de priorité

III.3.a. Protocole à priorité héritée

Le *protocole à priorité héritée*[2] ou *PPH* [SRL90] offre une solution au problème de l'inversion de priorité. Le principe de cet algorithme est de modifier la priorité de la tâche provoquant le blocage afin d'éviter que d'autres tâches de priorité intermédiaire ne préemptent la tâche détenant la ressource.
Voici, étape par étape, le principe de fonctionnement du protocole à priorité héritée :

1. Si la tâche τ_i essaie de prendre une ressource R détenue par une tâche τ_j de priorité inférieure, alors τ_i est bloquée ;
2. τ_j va "hériter" de la priorité de τ_i, tant que τ_j détiendra la ressource demandée par τ_i ;
3. Lorsque τ_j libère la ressource R, la tâche de plus forte priorité demandant la ressource R est activée. Si la tâche τ_j bloquait une autre tâche τ_k due à une autre ressource R', alors τ_j hérite de la priorité de τ_k, sinon, τ_j retrouve sa priorité initiale ;
4. L'héritage de priorité est transitif : si τ_k bloque τ_j qui bloque τ_i, alors la tâche τ_k hérite de la priorité de τ_i.

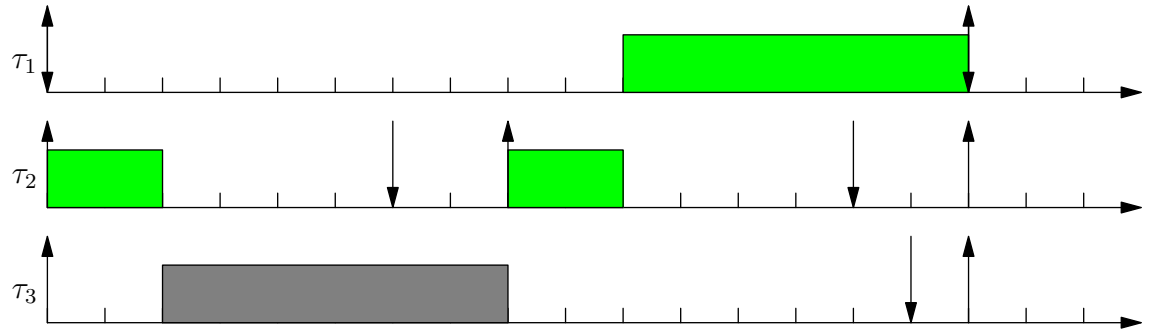
Malheureusement, ce protocole ne permet pas d'éviter les interblocages.

III.3.b. Protocole à priorité plafond

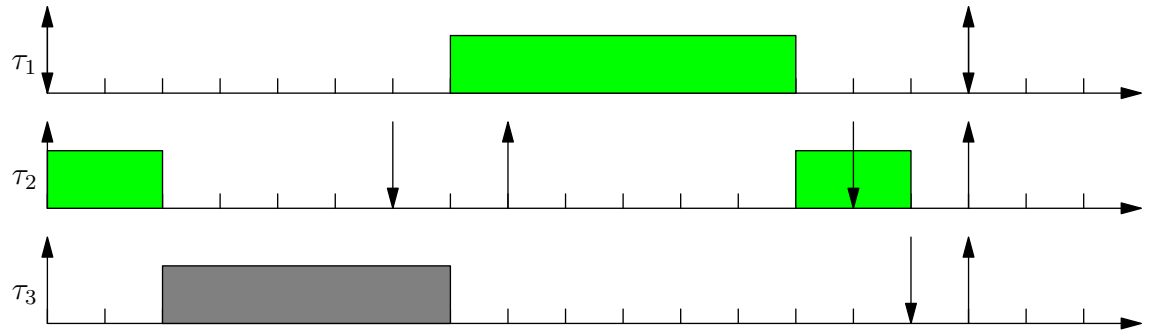
Le *protocole à priorité plafond*[3] ou *PPP* reprend les avantages du protocole à priorité héritée, en évitant les inversions de priorité, et évite, de plus, les interblocages. Ce protocole a été introduit par Sha, Rajkumar et Lehoczky dans [SRL90].

1. Chaque ressource R_k est associée à une priorité plafond $C(R_k)$ égale à la priorité de la tâche la plus prioritaire pouvant en faire la demande ;
2. Soit R^* la ressource ayant la plus forte priorité plafond parmi toutes les ressources verrouillées.

2. ou *Priority Inheritance Protocol* en anglais
3. ou *Priority Ceiling Protocol* en anglais



$C_3 = 6$, toutes les échéances sont respectées



$C_3 = 5$, τ_2 rate son échéance

FIGURE 1.8 – Anomalie d'ordonnancement

3. Supposons maintenant que la tâche de plus haute priorité prête à entrer en section critique soit τ_i. τ_i ne pourra entrer en section critique que si sa priorité est strictement supérieure à la priorité plafond $C(R^*)$. Dans le cas contraire, la tâche τ_i est mise en attente.
4. Lorsqu'une tâche est bloquée sur une ressource, elle transmet sa priorité à la tâche possédant la ressource
5. la transitivité définie dans le protocole à priorité héritée est conservée.

III.3.c. Protocole à priorité plafond immédiat

Ce protocole a été introduit par Kaiser dans [Kai82]. Il présente les mêmes avantages que le protocole à priorité plafond, tout en étant plus simple d'un point de vue de l'implémentation. Le principe du protocole à priorité plafond immédiat est résumé ci-dessous :

1. Chaque ressource R_k est associée à une priorité plafond $C(R_k)$ égale à la priorité de la tâche la plus prioritaire susceptible de l'utiliser (comme pour PPP) ;

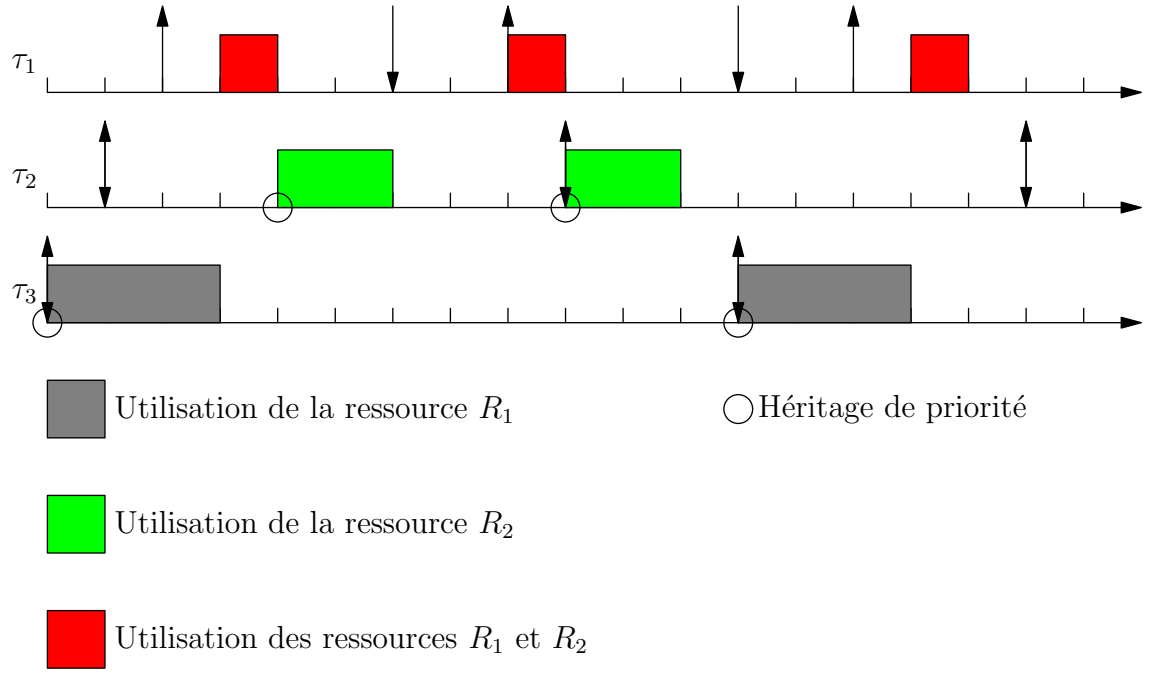


FIGURE 1.9 – Illustration du protocole à priorité plafond immédiat

2. Soit R^* la ressource ayant la plus forte priorité plafond parmi toutes les ressources verrouillées (comme pour PPP).
3. Une tâche ne peut prendre la ressource (c'est-à-dire entrer en section critique), que si sa priorité est strictement supérieure à R^* ou si elle détient elle-même le plafond système (comme pour PPP)
4. Lorsqu'une tâche entre en section critique en prenant la ressource R_k, elle hérite de la priorité plafond de la ressource R_k durant toute la durée de sa section critique.

Exemple Un exemple d'ordonnancement utilisant le protocole à priorité plafond immédiat est fourni figure 1.9. Le système est composé :
– de deux ressources, R_1 et R_2 ;
– de la tâche τ_1, nécessitant les ressources R_1 et R_2 ;
– de la tâche τ_2, nécessitant la ressource R_2 ;
– de la tâche τ_3, nécessitant la ressource R_3.
Les deux ressources R_1 et R_2 ont la même priorité plafond 0, puisque toutes deux requises par la tâche de plus grande priorité. A l'instant $t = 0$, la tâche τ_3 est la seule à être prête et s'exécute donc, en prenant la ressource R_1. La tâche τ_3 hérite donc de la priorité plafond de R_1, à savoir 0.
A l'instant $t = 1$, τ_2 se réveille. τ_2 est normalement plus prioritaire que τ_3,mais la prise de la ressource R_1 par la tâche τ_3 a modifié sa priorité. τ_3 est donc plus prioritaire que τ_2 et c'est

donc τ_3 qui s'exécute.
A l'instant $t = 2$, τ_1 se réveille. La ressource critique R_1 étant déjà prise, τ_1 est bloquée.
A l'instant $t = 3$, τ_3 vient de terminer son exécution et a libéré la ressource R_1. τ_1 et τ_2 sont prêtes, et comme τ_1 est plus prioritaire que τ_2, τ_1 s'exécute et entre en section critique, par la prise des ressources R_1 et R_2.
A l'instant $t = 3$, τ_1 termine son exécution, libérant les ressources R_1 et R_2. τ_2 peut maintenant s'exécuter, tout en héritant de la priorité plafond de la ressource R_2.
A l'instant $t = 5$, τ_2 termine son exécution, libérant la ressource R_2 et retrouvant sa priorité initiale.

III.4. Contraintes de précédence

En plus de partager des ressources, les tâches peuvent communiquer entre elles afin de s'échanger des informations. Ces échanges d'information constituent des points de synchronisation.
Bien souvent, les tâches sont découpées selon leurs points de synchronisation, ce qui permet de faire les hypothèses suivantes :
– une tâche ne peut recevoir un message qu'au début de son exécution ;
– une tâche ne peut envoyer de messages qu'à la fin de son exécution.
Ce découpage des tâches selon les points de synchronisation introduit des relations de précédence entre une tâche émettrice τ_e et une tâche réceptrice τ_r. Une telle relation, nommée *contrainte de précédence*, sera notée $\tau_e \prec \tau_r$. Il est à noter que cette relation définie un ordre partiel [SS94]. De cette relation découle également une contrainte sur les périodes : la période de la tâche émettrice et de la tâche réceptrice doivent être égale (i.e., $T_e = T_r$).

III.4.a. Gestion des contraintes de précédence avec EDF

Lorsqu'EDF est utilisé pour ordonnancer un système de tâches, il est possible de s'assurer du respect de l'ordre partiel en jouant sur les paramètres des tâches, et plus particulièrement sur les dates d'arrivée et sur les échéances relatives. Ainsi, pour que la relation $\tau_i \prec \tau_j$ soit respectée, il suffit de s'assurer que :
– $r_i + C_i \leq d_i$, sans quoi, il serait strictement impossible pour la tâche τ_i de respecter son échéance
– $d_i \leq d_j - C_j$, ce qui permet de s'assurer que l'émetteur d'un message ait une échéance qui arrive plus tôt que le destinataire, et que cette échéance laisse ensuite le temps au récepteur du message de compléter son exécution.

III.4.b. Gestion des contraintes de précédence en priorité fixe

La gestion des contraintes de précédence en priorité fixe est un peu plus délicate :
– soit le modèle de tâche incorpore un protocole de synchronisation permettant de s'assurer du respect des relations de précédence ;

	C_i	$T_i = D_i$	π_i
τ_1	5	18	2
τ_2	2	6	1
τ_3	1	18	3
τ_4	5	18	0

TABLE 1.3 – Système de tâche illustrant les anomalies d'ordonnancement en présence de contraintes de précédence en priorité fixe

– soit il faut s'assurer "manuellement" que le successeur d'une tâche ne puisse commencer son exécution qu'une fois que ses prédécesseurs aient terminés la leur. Pour cela, il faut modifier la date d'arrivée du successeur d'une tâche d'une durée correspondant au pire temps de réponse de tous ses prédécesseurs [Tin94].

A noter également que les contraintes de précédence en priorité fixe peuvent être la source d'anomalies d'ordonnancement, comme le montre la figure 1.10, qui est une représentation graphique du système de tâches de la table 1.3. Une réduction du temps d'exécution de la tâche τ_1 de 2 unités de temps va conduire la tâche τ_2 à rater son échéance. En effet, la tâche τ_4, qui est la tâche la plus prioritaire du système de tâches, doit attendre que son prédécesseur, la tâche τ_3, se soit terminée pour commencer son exécution. Le fait que τ_1 termine son exécution plus tôt permet à τ_3 de s'exécuter plus tôt. Ainsi, à l'instant $t = 6$, les tâches τ_2 et τ_4 sont prêtes à être exécutées, alors qu'auparavant, seule la tâche τ_2 était prête. Comme τ_4 est la tâche la plus prioritaire, c'est elle qui s'exécute, au détriment de τ_2.

IV. Ordonnancement multiprocesseur

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

– le problème de *placement des tâches* : sur quel(s) processeur(s) une tâche va-t-elle s'exécuter ?
– le problème de la *migration des tâches* : une tâche peut-elle changer de processeur pour s'exécuter ?
– le problème de la *gestion des ressources* : une ressource ne doit pas pouvoir être prise par deux tâches en même temps s'exécutant sur deux processeurs différents ;
– le problème de l'*ordonnancement des tâches* : affectation des priorités.

Dans le cadre de cette thèse, nous nous intéresserons uniquement au problème de placement, de migration et d'ordonnancement des tâches, sans prendre en compte la gestion des ressources (nous supposerons que les tâches sont indépendantes).

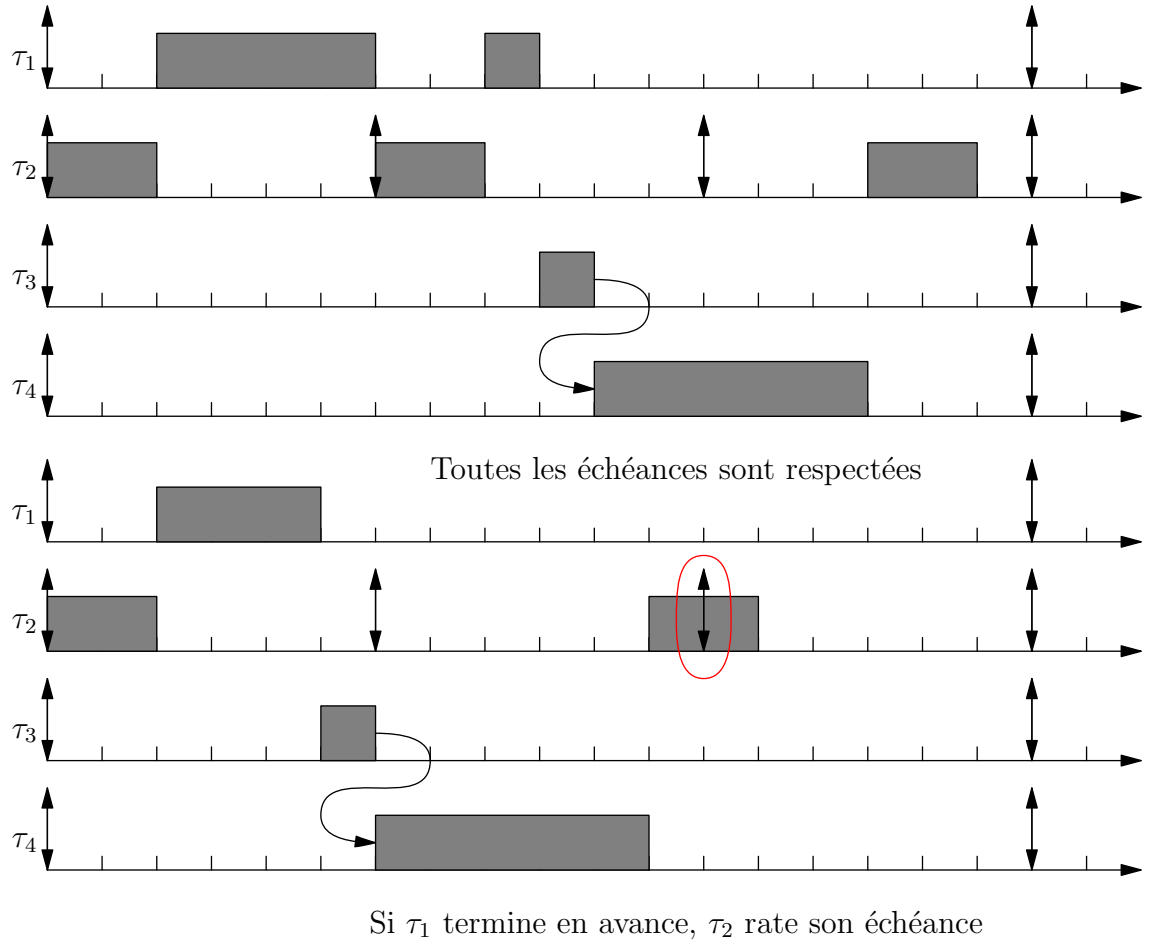


FIGURE 1.10 – Exemple d'anomalie d'ordonnancement en présence de contraintes de précédence en priorité fixe

IV.1. Migrations des tâches

Il existe trois principales catégories de tâches, en fonction du type de migrations autorisées :

- *sans migration* : lorsqu'une tâche est assignée à un processeur, toutes les instances de cette tâche s'exécuteront sur ce processeur ;
- *avec migration des tâches* : les tâches peuvent migrer, c'est-à-dire qu'une instance d'une tâche peut s'exécuter sur un processeur et l'instance suivante sur un autre processeur. Toutefois, une instance ayant commencée son exécution sur un processeur donné ne pourra pas migrer et fera donc toute son exécution sur le même processeur.
- *avec migration des instantes* : l'exécution d'une instance d'une tâche peut être suspendue, pour être reprise plus tard sur un autre processeur.

IV.2. Classification

Les algorithmes d'ordonnancement peuvent être classés dans différentes catégories, en fonction de leurs caractéristiques :
- *stratégie globale* : sur un système comprenant m processeurs, un algorithme d'ordonnancement utilisant une stratégie globale va affecter les m tâches les plus prioritaires aux m processeurs.
- *stratégie par partitionnement* : le principe d'un algorithme utilisant une stratégie par partitionnement est de placer chaque tâche sur un processeur, et ensuite d'exécuter sur chaque processeur un algorithme d'ordonnancement monoprocesseur.

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

IV.3. Optimalité

Un résultat intéressant démontré par Hong et Leung est le suivant :

Théorème 11 ([HL92]). *Il n'existe pas d'algorithme en-ligne optimal pour des systèmes multiprocesseurs.*

Toutefois, lorsqu'on restreint le cadre d'étude et que l'on ne considère uniquement des systèmes de tâches périodiques, alors il existe des algorithmes optimaux, comme les algorithmes de type Pfair (cf. section IV.5.a).

IV.4. Algorithme d'ordonnancement utilisant une stratégie par partitionnement

IV.4.a. Généralité

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

IV.4.b. First-Fit et Best-Fit

Parmi les heuristiques existantes pour résoudre ce type de problème, nous pouvons citer les algorithmes de type *First Fit Decreasing* et *Best Fit Decreasing*, qui reposent tous deux sur le

même principe : on trie la liste des tâches (par exemple, par charge décroissante), et on affecte chaque tâche dans l'ordre à un processeur, selon un critère d'acceptation.
La différence entre les deux types d'algorithmes réside sur la manière dont est placée chaque tâche :
- pour les algorithmes de type *First Fit Decreasing*, la tâche est placée sur le premier processeur répondant aux critères ;
- pour les algorithmes de type *Best Fit Decreasing*, la tâche est placée sur le processeur le mieux rempli qui puisse l'accepter.

Il est possible d'avoir de nombreuses variantes, en modifiant l'ordre de tri et / ou le critère d'acceptation :
- FBB-FFD (*Fisher, Baruah and Baker - First Fit Decreasing*) décrit dans [FBB06]
- RMNF (*Rate Monotonic First-Fit*) décrit dans [DL78]
- RMFF (*Rate Monotonic Next-Fit*) décrit dans [DL78]
- RMST (*Rate Monotonic Small Task*) décrit dans [BLOS95]
- RMGT (*Rate Monotonic General Task*) décrit dans [BLOS95]

Bien entendu, la liste ci-dessus n'est pas exhaustive. De même que les algorithmes de type *First Fit* ou *Best Fit* ne sont pas les seuls existants.

IV.4.c. Condition suffisante d'ordonnançabilité : borne de 50%

Un moyen pratique et rapide de déterminer si un système de tâches est ordonnançable est de se baser sur le facteur d'utilisation du système.

Théorème 12 ([LGDG00]). *Si un système de tâches à échéance sur requête vérifie :*

$$U \leq 0.5(m+1) \tag{1.7}$$

où m est le nombre de processeurs alors il existe un algorithme d'ordonnancement utilisant une stratégie par partitionnement pouvant ordonnancer fiablement ce système.

Cette borne est une borne serrée, puisqu'il est toujours possible de construire un système de tâche ayant une charge légèrement supérieure à 50% par processeur qui ne soit pas ordonnançable.

IV.5. Algorithme d'ordonnancement utilisant une stratégie globale

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

IV.5.a. Algorithme de type Pfair

Les algorithmes de type Pfair ont la particularité d'exécuter les tâches à un taux régulier. En effet, si, pour des algorithmes d'ordonnancement classique, le taux d'exécution de la tâche τ_i

avoisine u_i lorsqu'on considère de grands intervalles, ce n'est pas le cas pour des intervalles petits et le taux peut varier de manière relativement importante. Un algorithme Pfair a cette caractéristique de limiter les variations de ce taux en s'assurant que le taux d'exécution de la tâche τ_i reste voisin de u_i, quelle que soit la longueur de l'intervalle considéré.
De manière plus formelle, si on dénote $S(\tau_i, t)$ le nombre d'unités processeur utilisées par la tâche τ_i dans l'intervalle $[0, t[$, alors on peut définir la fonction retard par :

$$\text{retard}(\tau_i, t) = u_i \cdot t - S(\tau_i, t) \tag{1.8}$$

Un ordonnancement sera alors dit Pfair si, et seulement si, la condition suivante est vérifiée :

$$-1 < \text{retard}(\tau_i, t) < 1 \quad \forall t, \tau_i \in \tau \tag{1.9}$$

L'intérêt majeur des algorithmes de type Pfair est qu'ils sont optimaux dans le cadre résumé par le théorème 13 :

Théorème 13. *L'algorithme Pfair est optimal dans le cadre d'un système de tâches indépendantes à échéance sur requête et à départ simultané, lorsque la migration des instances est autorisée [BCPV96].*

De plus, Baruah a donné une condition nécessaire et suffisante d'ordonnançabilité pour ce type d'algorithme :

Théorème 14 ([BCPV96])**.** *Un système τ de tâches indépendantes, synchrones et à échéance sur requête est ordonnançable en utilisant un algorithme de type Pfair sur m processeurs identiques, si et seulement si :*

$$U \leq m \tag{1.10}$$

Parmi les algorithmes d'ordonnancement Pfair, nous pouvons citer :
– PF [BCPV96]
– PD [BGP95]
– PD^2 [AS00]
Ces trois algorithmes affectent une priorité aux instances des tâches en utilisant une stratégie proche de celle utilisée par EDF, seule diffère la manière de lever les ambiguïtés, c'est-à-dire dans le cas d'égalité de priorité.

IV.5.b. g-EDF

L'algorithme EDF peut être utilisé dans un cadre multiprocesseur et est alors désigné sous le nom *g-EDF* (pour *global EDF*). L'affectation des priorités se fait comme dans le cas monoprocesseur, mais au lieu d'exécuter la tâche la plus prioritaire, les m tâches les plus prioritaires sont exécutées, où m correspond au nombre de processeurs disponibles.
EDF étant un algorithme bien connu en monoprocesseur aux caractéristiques remarquables, il a reçu beaucoup d'attention en multiprocesseur [GFB03, Bak05b, PHK$^+$05, BB06, BB08].

IV.5.c. EDZL

L'algorithme *EDZL* [CLAL02] (pour *Earliest Deadline Zero Laxity*) est une variante de l'algorithme EDF. L'affectation des priorités se fait de la même manière, mais si, à un instant donné, une tâche a une laxité nulle, alors cette tâche est affectée de la plus grande priorité.
L'intérêt majeur de l'algorithme EDZL est résumé par le théorème 15 :

Théorème 15 ([PHK+05]). *L'algorithme EDZL domine g-EDF.*

L'algorithme EDZL a fait l'objet d'études et d'analyses dans la littérature [PHK+05, BCB08].

V. Ordonnancement distribué

L'ordonnancement des systèmes distribués se différencie de l'ordonnancement monoprocesseur du fait de la présence non plus d'un mais de plusieurs processeurs ainsi que de la présence de réseaux permettant à des tâches s'exécutant sur des processeurs différents de communiquer entre elles grâce à l'échange de messages.

V.1. Problématique

Une telle architecture pose de nombreux problèmes qui n'étaient pas présents dans le cas de l'ordonnancement monoprocesseur. Il reprends les problèmes supplémentaires posés par le cas multiprocesseur, et en introduit de nouveaux :
- le problème de la *synchronisation* : éviter la dérive des horloges entre les différents processeurs ;
- le problème de l'*ordonnancement des messages* : affecter une priorité aux messages sur les réseaux.

Dans la suite de cette thèse, nous considérerons principalement les problèmes liés à l'ordonnancement des tâches et des messages. De plus, nous supposerons qu'aucune migration de tâches n'est autorisée.

V.2. Ordonnancement des tâches et des messages

Comme les tâches en monoprocesseur qui se partagent une ressource commune (le processeur), les messages qui transitent se partagent le réseau. Du fait de cette analogie, les messages sont souvent modélisés comme des tâches non préemptibles héritant des caractéristiques de leur expéditeur et destinataire :
- la période du message correspond à la période de la tâche émettrice ;
- la durée d'exécution du message, correspond à la durée de transit du message sur le réseau ;
- l'échéance du message est déterminée à partir de l'échéance de la tâche réceptrice.

Une hypothèse souvent faite, est qu'une tâche ne peut recevoir un message qu'au début de son exécution, et ne peux en envoyer un qu'à la fin de son exécution. Cette hypothèse, que nous faisons dans le cadre de cette thèse, implique les contraintes de précédence suivantes :
- un message ne peut être envoyé par une tâche qu'une fois que cette tâche est terminée.
- une tâche recevant un message ne peut débuter son exécution qu'à la réception de ce message (elle est donc soumise à une gigue sur activation).

Il existe de nombreux travaux traitant de ce type de problème, notamment autour de l'*analyse holistique* [Tin94] permettant la validation de systèmes distribués et qui sera présentée dans le chapitre suivant. Nous pouvons également citer les travaux de Richard [Ric02] traitant directement du problème, ou encore ceux de Hladik [Hla04] utilisant la notion de contraintes pour représenter les communications.

Chapitre

2 Validation des systèmes temps réel

Sommaire

Résumé

Ce chapitre est une introduction à l'analyse holistique, la méthode de validation qui sera utilisé tout au long du chapitre 3.

I. Introduction

La *validation temporelle* des systèmes temps réel est nécessaire afin de prouver leur validité, c'est-à-dire de s'assurer que chaque tâche respectera son échéance.
Pour se faire, plusieurs possibilités s'offrent à nous :
– la *simulation* ;
– l'*évaluation de la demande processeur* ;
– le *calcul du pire temps de réponse.*
Nous allons détailler chacune des méthodes dans le cadre de ce chapitre. Toutefois, ces méthodes ne peuvent donner de résultats fiables qu'au travers l'analyse d'un scénario dit *pire cas*, résultant de l'analyse des caractéristiques des tâches. L'étude de ces scénarios pire cas fait l'objet de la section III.
Dans le cadre de ce chapitre, lorsque nous considérerons des systèmes de tâches à priorité fixe, nous supposerons que les tâches sont triées par priorité, c'est-à-dire que $i < j$ implique que la tâche τ_i est plus prioritaire que la tâche τ_j (i.e., $\pi_i < \pi_j$).

II. Simulation

Le principe de la simulation est de simuler le comportement d'un système afin de détecter les fautes temporelles. Bien qu'un ordonnancement soit par définition infini, il est possible de simuler un système de tâches à la recherche de fautes temporelles sur un intervalle de temps fini. Ceci est réalisable grâce au caractère périodique des tâches composant le système. Cet intervalle d'étude est appelé *intervalle de faisabilité*[1] [LM80].
La détermination de la longueur de l'intervalle de faisabilité dépend des caractéristiques du système de tâches et se base sur la notion d'*hyperpériode*, que l'on pourrait désigner comme étant la période du système.

Définition 10. *L'hyperpériode d'un système de tâche est égale au PPCM*[2] *des périodes des tâches du système.*

II.1. Système de tâches synchrones

Dans le cadre d'un système de tâches synchrones, la longueur de l'intervalle de faisabilité est donnée par le théorème 16 :

Théorème 16 ([LM80])**.** *Dans la cas d'un système de tâches synchrones, l'intervalle de faisabilité est* $[0, PPCM(T_i)[$.

1. ou *feasibility interval* en anglais
2. plus petit commun multiple

II.2. Système de tâches asynchrones

Dans le cadre plus général d'un système de tâches asynchrones, la durée de l'intervalle de faisabilité est allongée :

Théorème 17 ([LM80]). *Dans la cas d'un système de tâches asynchrones, l'intervalle de faisabilité est* $[0, \max r_i + 2PPCM(T_i)[$.

La présence d'anomalies d'ordonnancement rend cette méthode de validation inutilisable en pratique puisque cette approche suppose que le pire comportement des tâches ne survient que lorsque celles-ci s'exécutent avec leur pire durée d'exécution. En effet, simuler le comportement du système en se basant sur le comportement pire cas des tâches ne conduit pas obligatoirement au pire comportement du système, et les variations des paramètres des tâches (notamment, les temps d'exécution) durant la vie du système peuvent conduire à des situations qui n'auront jamais été simulées.
Afin d'éviter ce problème d'une part, et de limiter la durée de l'intervalle d'étude d'autre part, il est nécessaire de caractériser le pire scénario d'activation des tâches pouvant survenir dans l'ordonnancement. Sur la base de ce scénario (ou ensemble de scénarios), une analyse d'ordonnançabilité est effectuée.

III. Le scénario pire cas

III.1. Définition et objectif

Le scénario pire cas est le scénario qui sera considéré lors de la validation d'un système. L'intérêt majeur de ce scénario est qu'il est, comme l'indique son nom, pire cas, c'est-à-dire le scénario pour lequel les tâches auront leur plus grand temps de réponse.
L'objectif est de déterminer et valider ce scénario pire cas, puisque la validation de ce scénario garantira la validation de scénarios où les temps de réponse des tâches sont plus faibles, et validera ainsi le système, quel que soit le scénario considéré.

III.2. Instant critique

La notion d'instant critique est une notion importante dans la détermination du scénario pire cas :

Définition 11 ([LL73]). *Un instant critique correspond à un instant où toutes les tâches du systèmes sont activées simultanément.*

En effet, de nombreux scénarios pire cas sont initiés par un instant critique, comme nous le verrons dans les sections suivantes.

III.3. Scénario pire cas dans le cadre d'un ordonnancement à priorité fixe

III.3.a. Tâches indépendantes

Dans un cadre monoprocesseur et en priorité fixe pour des tâches périodiques et à échéance arbitraire, des travaux ont montré que le pire temps de réponse pour une tâche τ_i est obtenu dans le cadre suivant :

Théorème 18 ([LL73, Mok83, JP86]). *Le pire temps de réponse pour une tâche τ_i est obtenu lorsque toutes les tâches sont activées à leur rythme maximal et à un instant critique.*

Liu et Layland ont montré dans [LL73] que le pire temps de réponse d'une tâche τ_i correspondait au temps de réponse de la première instance de la tâche τ_i, pour des tâches à échéance sur requête. Joseph et Pandya ont étendu ce résultat pour des tâches à échéance contrainte dans [JP86].
Enfin, Lehoczky a montré que dans un cadre où les échéances étaient arbitraires, le pire temps de réponse d'une tâche τ_i n'était plus forcément associé à sa première instance, mais pouvait l'être à une instance ultérieure. Afin d'étudier au mieux ce comportement, Lehoczky a introduit la notion de *période d'activité de niveau i* [3], où le processeur n'exécute que des tâches τ_j dont la priorité est supérieure ou égale à celle de τ_i (i.e., $\pi_j < \pi_i$).
Lehoczky a introduit cette notion dans le but de déterminer la longueur de l'intervalle d'étude lors de la détermination du temps de réponse d'une tâche τ_i :

Théorème 19 ([Leh90]). *Le pire temps de réponse d'une tâche τ_i survient durant une période d'activité de niveau i où toutes les tâches débutent par un instant critique.*

Le principe pour déterminer le pire temps de réponse d'une tâche est alors :

1. déterminer la longueur L de la période d'activité de niveau i ;
2. calculer le temps de réponse de toutes les instances de τ_i s'exécutant durant cette période d'activité de longueur L ;
3. le pire temps de réponse de la tâche τ_i est alors obtenu en prenant le maximum des temps des réponses des instances de la tâche τ_i.

III.3.b. Prise en compte de la gigue sur activation

Un inconvénient du scénario que nous avons présenté jusqu'à présent est que nous faisons l'hypothèse suivante : nous supposons que la date d'activation de chaque tâche est connue *a priori*. Or, ceci n'est pas tout le temps vrai si la tâche est bloquée, comme dans le cas, par exemple, de l'attente d'un message. Ce problème porte le nom du problème de la gigue sur activation [4] (cf. figure 2.1).

3. ou *level-i busy period* en anglais
4. ou *release jitter problem* en anglais

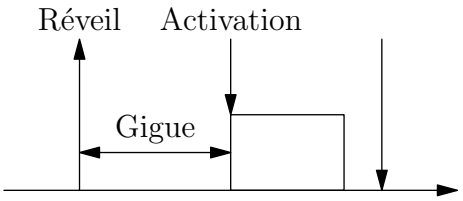


FIGURE 2.1 – Gigue sur activation

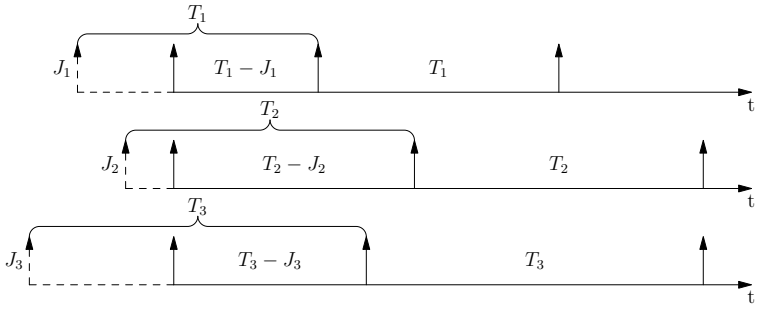


FIGURE 2.2 – Scénario pire cas en présence de gigues sur activation

Il est à noter également que la gigue peut être utilisée pour modéliser les délais d'activation d'une tâche associés aux routines du noyau.
Une des conséquences de ce problème est que la différence entre deux activations successives de deux instances d'une tâche τ_i peut être plus grande que la différence entre les deux réveils successifs correspondant. La gigue d'une tâche τ_i, que nous désignerons par J_i, doit alors être prise en compte lors du calcul du temps de réponse de la tâche τ_i.
La prise en compte de la gigue modifie un peu le scénario pire cas. La plus longue période d'activation se produit lorsque les tâches s'activent en même temps après avoir été retardées du maximum de leur gigue (cf. figure 2.2).

III.3.c. Prise en compte des ressources

La prise en compte des ressources ne modifie pas la caractérisation du scénario pire cas. Par contre, elle modifiera la manière de calculer les temps de réponse, comme nous le verrons dans le paragraphe V.2.

III.4. Scénario pire cas dans le cadre d'un ordonnancement EDF

Il n'existe malheureusement pas, à notre connaissance, de caractérisation d'un unique scénario pire cas dans le cas des algorithmes à priorité dynamique en général, et dans le cas de EDF en particulier, en comparaison des priorités fixes. En effet, Spuri a montré dans [Spu96] que le pire temps de réponse d'une tâche ne survient pas nécessairement durant la première période

d'activité dans le cadre d'un ordonnancement par EDF, contrairement aux priorités fixes. Dans ce cas, la détermination du scénario pire cas se fait en énumérant un ensemble de scénarios menant potentiellement au scénario pire cas. Spuri, en montrant le théorème 20, a caractérisé les scénarios candidats :

Théorème 20 ([Spu96]). *Le pire temps de réponse d'une tâche sporadique τ_i est trouvé dans une période d'activité où toutes les tâches autre que τ_i sont réveillées de façon synchrone et à leur rythme maximum (cas des tâches sporadiques).*

IV. Evaluation de la demande processeur

IV.1. Ordonnancement à priorité fixe

L'évaluation de la demande processeur se base sur un comptage du nombre de requêtes activées durant une période d'activité.

Définition 12. *La demande processeur des tâches réveillées avant une date donnée t est notée $rbf(\tau_i, t)$ (Request Bound Function) :*

$$rbf(\tau_i, t) \stackrel{\text{def}}{=} \max\left(0, \left\lceil \frac{t}{T_i} \right\rceil\right) C_i \tag{2.1}$$

Disposant de la demande processeur d'une tâche, il est aisé de généraliser pour toutes les tâches d'un système et d'introduire la fonction de travail du processeur $W(t)$:

$$W(t) = \sum_{i=1}^{n} rbf(\tau_i, t) \tag{2.2}$$

Une variante de cette dernière fonction, ne prenant en compte que les tâches de priorité supérieure ou égale à i est notée par :

$$W_i(t) = \sum_{j=1}^{i} rbf(\tau_j, t) \tag{2.3}$$

Le test d'ordonnançabilité est alors donné par le théorème suivant :

Théorème 21 ([LSD89]). *Pour un système de tâches à départ simultané, une tâche τ_i sera ordonnançable si, et seulement si, il existe un instant $t \in [0, D_i[$ tel que :*

$$W_i(t) \leq t \tag{2.4}$$

Il reste un dernier souci à résoudre. Le théorème est valable à condition de tester toutes les valeurs de t dans l'intervalle $[0, D_i[$. Heureusement pour nous, la fonction rbf est une fonction en escalier, et il suffit donc de tester un nombre fini de valeurs pour t. Ces points s'appellent des points d'ordonnancement [5], et l'ensemble S_i des points d'ordonnancement a été initialement défini par [LSD89] :

$$S_i \stackrel{\text{def}}{=} \left\{ kT_j | j = 1, \ldots, i, k = 1, \ldots \left\lfloor \frac{D_i}{T_j} \right\rfloor \right\} \tag{2.5}$$

Des travaux plus récents ont également permis de réduire cet ensemble de points d'ordonnancement. On peut citer notamment les travaux de Manabe [MA98] ou de Bini et Buttazzo [BB04b].

IV.2. Ordonnancement EDF

En priorité dynamique, la validation d'un système pour un ordonnancement EDF repose sur la fonction suivante :

Définition 13. *La demande processeur des tâches ayant leur échéance avant un instant t est notée $dbf(t)$ (Demand Bound Function) :*

$$dbf(t) \stackrel{\text{def}}{=} \max\left(0, \left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) C_i \tag{2.6}$$

Le test d'ordonnançabilité repose alors sur le théorème 22 :

Théorème 22 ([BRH90])**.** *Un système de tâches périodiques et synchrones est ordonnançable si, et seulement si :*

$$dbf(t) \leq t \quad \forall t, 0 < t < t_{lim} \tag{2.7}$$

Il reste maintenant à déterminer la valeur de t_{lim}. De nombreux travaux ont été faits à ce sujet. On peut notamment citer ceux de Baruah [BRH90], qui a établit que :

Théorème 23. *Pour un système de tâches périodiques, à départ simultané et dont le facteur d'utilisation U est strictement inférieur à 1,*

$$t_{lim} = \min\left(PPCM(T_i), \frac{U}{1-U} \max_{i=1\ldots n}(T_i - D_i)\right) \tag{2.8}$$

Cette borne a été améliorée par la suite dans [RCM96], qui a limité l'étude à la première période d'activité dans le cas de tâches synchrones, ou encore dans [ZB09].
Enfin, de même qu'en priorité fixe, la fonction dbf est une fonction en escalier. Il n'est donc pas nécessaire de vérifier toutes les valeurs de t dans l'intervalle $[0, t_{lim}[$, mais seulement les points de discontinuités de la fonction dbf. Ce test a été montré Co-$\mathcal{NP}$-Difficile au sens faible récemment [ER10].

5. ou *scheduling points* en anglais

V. Calcul du pire temps de réponse

La dernière méthode de validation consiste à déterminer analytiquement le temps de réponse des différentes tâches composant un système, en se basant sur leurs caractéristiques, afin de vérifier que chaque tâche respectera son échéance.

V.1. Détermination du temps de réponse

Nous avons vu précédemment que, dans le cas où les échéances sont arbitraires, ce n'est pas obligatoirement le temps de réponse de la première instance d'une tâche τ_i qui donnera le pire temps de réponse de la tâche τ_i. Il est donc nécessaire de déterminer le temps de réponse de toutes les instances de la tâche τ_i durant la période d'activité correspondant au pire scénario d'activation des tâches pour déterminer le pire temps de réponse de τ_i.

V.1.a. Détermination du temps de réponse d'une instance donnée

Pour déterminer le temps de réponse d'une tâche τ_i à échéance arbitraire (cas le plus général), plusieurs éléments doivent être pris en compte :
- l'interférence due aux tâches plus prioritaires, c'est-à-dire la durée pendant laquelle la tâche est préemptée au profit de tâches plus prioritaires ;
- la charge processeur due aux instances de τ_i.

L'interférence $I_i(t)$ que subit τ_i due aux tâches plus prioritaires sur une période d'activité de longueur t est déterminée par la formule suivante :

$$I_i(t) = \sum_{j \in hp(i)} \left\lceil \frac{t}{T_j} \right\rceil C_j \tag{2.9}$$

où $hp(i)$ désigne l'ensemble des tâches plus prioritaires que τ_i.
Si, de plus, on suppose que durant cet intervalle de longueur t, q requêtes de la tâche τ_i sont activées, alors la charge processeur due aux instances de τ_i vaut :

$$qC_i \tag{2.10}$$

Le temps de réponse de la $q^{\text{ème}}$ instance de τ_i est déterminée par la résolution du système d'équations récursif suivant :

$$\begin{cases} R_{i,q}^{(0)} = qC_i \\ R_{i,q}^{(l+1)} = qC_i + I_i(R_{i,q}^{(l)}) \end{cases} \tag{2.11}$$

Le système est résolu lorsque le plus petit point fixe est atteint, c'est-à-dire lorsque $R_{i,q}^{(l)} = R_{i,q}^{(l+1)}$. Dans ce cas, le temps de réponse de la $q^{\text{ème}}$ instance de τ_i est donné par :

$$Tr_{i,q} = R_{i,q}^{(l)} - (q-1)T_i \tag{2.12}$$

Il est nécessaire de retrancher $(q-1)T_i$ pour obtenir le temps de réponse, puisque la première instance commence à l'instant 0, la deuxième à l'instant T_i, la troisième à l'instant $2T_i$, etc... et que le temps de réponse est, par définition, le temps s'écoulant entre le moment où l'instance est réveillée et sa terminaison.
La résolution de ce système a été montré $\mathcal{NP}$-Difficile au sens faible dans [ER08].

V.1.b. Détermination du pire temps de réponse d'une tâche

Disposant des temps de réponse des instances de la tâche τ_i, il est aisé de déterminer son pire temps de réponse Tr_i :

$$Tr_i = \max_{q=1,2,\ldots,Q}(Tr_{i,q}) \tag{2.13}$$

Le souci réside maintenant dans la détermination de Q, c'est-à-dire du nombre d'instances que l'on doit effectivement prendre en compte dans le calcul du temps de réponse.
Cette réponse nous est apportée par les travaux de Tindell :

Théorème 24 ([Tin94]). *La période d'activité de niveau i se termine lorsque le temps de réponse d'une instance de τ_i est inférieure à sa période T_i, c'est-à-dire lorsque :*

$$Tr_{i,q} \leq T_i \tag{2.14}$$

L'idée pour déterminer le temps de réponse de la tâche τ_i va être de considérer uniquement une instance de la tâche ($Q = 1$). Si le temps de réponse de cette instance est inférieure ou égale à la période T_i de τ_i, alors la fin de la période d'activité est atteinte, et le pire temps de réponse de τ_i correspond au temps de réponse de cette instance.
Dans le cas contraire, on prends en compte une instance supplémentaire de τ_i ($Q \leftarrow Q + 1$). La détermination du temps de réponse de cette nouvelle instance nous indique si la fin de la période d'activité est atteinte ou non. Si la fin de la période est atteinte, alors le temps de réponse de la tâche τ_i sera le maximum des temps de réponses de ses instances (cf. équation 2.13). Dans le cas contraire, on recommence la procédure en prenant de nouveau une instance supplémentaire.

V.2. Ressource

Jusqu'à présent, nous n'avons considéré que des tâches indépendantes. Nous allons étendre le résultat précédent afin de permettre d'y inclure la gestion des ressources.
La prise en compte des ressources s'effectue en ajoutant un terme B_i à l'équation 2.11, ce terme symbolisant le blocage dont peut souffrir une tâche qui est en attente d'une ressource déjà utilisée par une autre tâche :

$$R_{i,q}^{(l+1)} = qC_i + B_i + I_i(R_{i,q}^{(l)}) \tag{2.15}$$

Il est à noter qu'une tâche ne peut souffrir au plus qu'une seule fois d'un blocage dû à une tâche moins prioritaire, ce qui explique que le terme de blocage ne soit présent qu'une seule fois dans le calcul du pire temps de réponse.
Hormis pour l'équation 2.11, le reste des équations demeurre inchangé.

V.3. Gigue sur activation

En présence de gigues sur activation, l'instant critique survient lorsque toutes les tâches reçoivent leurs messages en même temps. Les dates d'activations sont donc décalées pour prendre en compte cette gigue sur activation. Ainsi, la date d'activation de la $q^{\text{ème}}$ instance devient :

$$\max((q-1)T_i - J_i, 0) \tag{2.16}$$

Tindell a montré dans [Tin94] que le théorème 24 restait valide en présence de gigues sur activation et donc que la période d'activité de niveau i se terminait lorsqu'une instance de τ_i avait un temps de réponse inférieur à sa période T_i. Toutefois, la démonstration a été démontrée comme étant incomplète dans [SL96] et un complément de preuve a été apporté dans [GGH98].
La gigue sur l'activation doit également être prise en compte lors du calcul de l'interférence des tâches plus prioritaires sur τ_i. L'équation 2.9 devient alors :

$$I_i(t) = \sum_{j \in hp(i)} \left\lceil \frac{J_j + t}{T_j} \right\rceil C_j \tag{2.17}$$

Ne reste plus que l'équation 2.12 à modifier pour tenir compte de la gigue sur l'activation :

$$Tr_{i,q} = R_{i,q}^{(l)} + J_i - (q-1)T_i \tag{2.18}$$

Le reste des équations est inchangé.

V.4. Récapitulatif

Pour déterminer le temps de réponse d'une tâche τ_i, en prenant en compte gigue sur activation et partage de ressources, le système d'équations à résoudre se résume donc, au final, à :

$$\begin{cases} R_{i,q} = qC_i + \displaystyle\sum_{j \in hp(i)} \left(\left\lceil \frac{J_j + R_{i,q}}{T_j} \right\rceil \right) C_j + B_i \\ Tr_i = \displaystyle\max_{q=1,\ldots,Q} (R_{i,q} + J_i - qT_i) \end{cases} \tag{2.19}$$

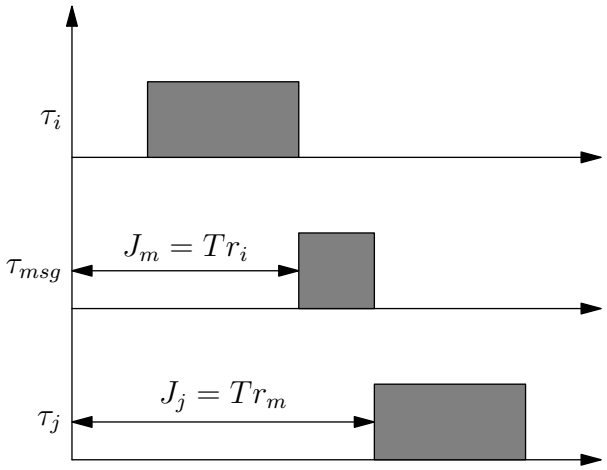


FIGURE 2.3 – Utilisation de la gigue sur activation pour la modélisation des dépendances

VI. Analyse holistique

VI.1. Principe de fonctionnement

L'analyse holistique est une technique de calcul de temps de réponse capable de s'appliquer à des systèmes distribués. Le principe de l'analyse holistique est de transformer le système distribué initial, en autant de systèmes monoprocesseurs qu'il y a de processeurs, permettant par la suite l'utilisation de techniques de calcul du temps de réponse sur chacun des sous-systèmes. Les messages, étant des entités relativement proches des tâches, seront modélisés comme des tâches. Ainsi, un message m sera noté τ_m.
Pour parvenir à ce but, les communications entre les tâches, et plus particulièrement les délais que peuvent induire l'attente d'un message à une tâche, seront modélisés à travers les gigues sur activation (figure 2.3). Le message τ_m ne pouvant être envoyé qu'une fois la tâche émettrice τ_i terminée, sa gigue J_m sera donc égale au temps de réponse de τ_i, c'est-à-dire Tr_i. De même, la tâche réceptrice τ_j, ne pouvant démarrer son exécution qu'à la réception du message τ_m, aura une gigue égale au temps de réponse du message τ_m, soit $J_j = Tr_m$.
Le problème est que pour connaître la date d'émission d'un message, il faut connaître le temps de réponse de la tâche émettrice. Et le calcul du temps de réponse de la tâche réceptrice ne pourra se faire qu'une fois le temps de réponse du message connu. Il existe donc une dépendance entre le temps de réponse des tâches et celui des messages. Ce qui n'est pas sans poser de problèmes lors du calcul des gigues, puisque le calcul des gigues dépend des temps de réponse et que les temps de réponse dépendent des gigues.
Une solution à ce problème est d'utiliser un algorithme itératif. Dans un premier temps, le temps de réponse des tâches et des messages est estimé. Puis dans un second temps, ce sont les gigues sur activation des tâches et des messages qui sont calculées. Grâce à ces nouvelles gigues, il est possible de mettre à jour les temps de réponse des tâches et des messages, qui

permettrons de mettre à jour les gigues et ainsi de suite. Le calcul itératif s'arrêtera lorsque tous les temps de réponse et toutes les gigues (que ce soit ceux des messages ou ceux des tâches), seront identiques d'une itération à une autre.
Le principe de calcul peut se résumer par les équations suivantes :

$$1 \leq i \leq n \begin{cases} Tr_i^{(0)} = C_i \\ J_i^{(0)} = 0 \\ Tr_i^{(j+1)} = \text{EvaluerTR}(J_i^{(j)}) \\ J_i^{(j+1)} = \max\limits_{k \in \Gamma^{-1}(i)} (Tr_k^{(j)}) \end{cases} \tag{2.20}$$

où $\Gamma^{-1}(i)$ correspond à l'ensemble des prédécesseurs de τ_i, c'est-à-dire que :
- si τ_i est un message, $\Gamma^{-1}(i)$ est un ensemble contenant une unique tâche, qui est la tâche émettrice du message τ_i ;
- si τ_i est une tâche, $\Gamma^{-1}(i)$ correspond à l'ensemble des messages que doit recevoir τ_i pour commencer son exécution.

Le calcul s'arrête lorsque $Tr_i^{(j+1)} = Tr_i^{(j)}$ et $J_i^{(j+1)} = J_i^{(j)}$.

VI.2. Hypothèses

Par la suite, nous allons considérer un système distribué, constitué d'un ensemble de machines reliées par un réseau, chaque machine étant équipée d'un unique processeur.
De plus, nous considérons les hypothèses suivantes :
- les horloges des différentes machines sont toutes synchronisées ;
- les messages ne sont lus qu'au début de l'exécution des tâches ;
- les messages ne sont envoyés qu'à la fin de l'exécution des tâches ;
- aucune migration de tâche n'est autorisée ;
- aucune ressource n'est partagée entre plusieurs processeurs (hormis le réseau) ;
- le réseau est fiable, c'est-à-dire que les communications se font sans erreurs et sont de durées bornées ;
- les tâches et les messages sont affectés d'une priorité fixe.

VI.3. Calcul du temps de réponse des tâches

De part le principe de l'analyse holistique, le calcul du temps de réponse des tâches se fait de la même manière qu'en monoprocesseur puisque les dépendances sont modélisées à travers les gigues sur activation. Il est donc possible d'utiliser l'équation 2.18 telle quelle.

VI.4. Calcul du temps de réponse des messages

Par la suite, nous ne considérerons que des réseaux de type CAN, même si les résultats peuvent être facilement étendu à d'autres types de réseaux, à condition de pouvoir déterminer le temps de propagation d'un message sur le type de réseau considéré.

VI.4.a. Principe

Pour déterminer le temps de réponse d'un message, nous allons considérer le message comme une tâche. Il reste alors à définir les caractéristiques du message :
- sa *période*, qui sera égale à la période de la tâche qui émet le message ;
- sa *gigue sur activation*, qui correspondra au temps de réponse de la tâche émettrice ;
- la *durée de propagation du message*, qui sera l'équivalent du temps d'exécution pour une tâche.

Ce faisant, nous considérerons alors le message comme une tâche, mais à caractère non préemptible.

VI.4.b. Durée de propagation

La détermination de la durée de propagation dépend de la longueur du message. Ainsi, il est nécessaire de connaître le type de réseau pour déterminer le temps de propagation.
Par exemple, si le réseau est un réseau CAN, le temps de propagation pour une trame de n octets, avec $n \leq 8$, est donné par la formule suivante :

$$C_m = \left(\left\lfloor \frac{34 + 8n}{4} \right\rfloor + 47 + 8n \right) \tau_{bit} \tag{2.21}$$

où τ_{bit} est la durée de propagation d'un bit.

VI.4.c. Prise en compte du caractère non préemptif

Disposant du temps de propagation, il ne reste plus qu'à prendre en compte le caractère non préemptif des messages. En réalité, le caractère préemptif ou non préemptif modifie peu l'ordonnancement :
- la charge du système reste identique ;
- lorsqu'on considère des algorithmes *conservatifs*, c'est-à-dire ne laissant jamais le processeur oisif s'il y a au moins une tâche en attente d'exécution, la position des temps creux reste la même.

La différence majeure survient lors de l'arrivée d'une tâche τ_i plus prioritaire que la tâche τ_j en cours d'exécution : au lieu de préempter la tâche τ_j pour permettre à τ_i de s'exécuter comme dans le cas préemptif, la tâche τ_j continue son exécution, et ce n'est qu'une fois celle-ci terminée que la tâche τ_i pourra s'exécuter.

Nous constatons donc que dans le cas non préemptif, la tâche plus prioritaire τ_i subit une attente due à l'exécution de la tâche moins prioritaire τ_j.
Pour déterminer le scénario pire cas dans cette situation, il faut modifier le scénario utilisé actuellement. En effet, si nous supposons que la tâche plus prioritaire τ_i et la tâche moins prioritaire τ_j arrive au même instant, puisque τ_j n'aura pas commencé son exécution, τ_i peut s'exécuter sans souffrir de délai dû à l'exécution de τ_j. Maintenant, imaginons que la tâche τ_j arrive juste avant τ_i. Dans ce cas, au moment de l'arrivée de la tâche τ_i, la tâche τ_j sera en cours d'exécution et la tâche τ_i devra donc attendre la terminaison de la tâche τ_j pour pouvoir s'exécuter. De plus, dans cette configuration, la tâche τ_i subira le blocage maximum que peut lui faire subir la tâche τ_j, qui correspond dans notre cas au pire temps d'exécution de τ_j.
Ces constations nous permettent d'introduire le théorème suivant caractérisant le scénario pire cas :

Théorème 25 ([Tin94])**.** *En considérant un ordonnancement non préemptif à priorité fixe, le pire temps de réponse pour une tâche τ_i survient durant la période d'activité de niveau i, lorsque toutes les tâches plus prioritaires sont réveillées à un instant critique, et en réveillant la tâche ayant le plus grand temps d'exécution parmi les tâches moins prioritaires juste avant l'instant critique.*

Tenant compte de cela, le calcul du temps de réponse de la $q^{\text{ème}}$ instance d'un message se déroule de manière similaire que pour une tâche :

$$Tr_{m,q} = J_m + \omega_m(q) - (q-1)T_m + C_m \tag{2.22}$$

où :
- J_m correspond à la *gigue sur activation* que subit le message, qui est égale au temps de réponse de la tâche émettrice du message τ_m ;
- $\omega_m(q)$ correspond au délai maximum pendant lequel la $q^{\text{ème}}$ instance du message τ_m peut être mis en attente, à cause de l'interférence due aux messages plus prioritaires et aux instances précédentes de τ_m, ainsi que le blocage que peut subir le message due à l'envoi d'un message moins prioritaire ;
- comme pour les tâches, il est nécessaire de retrancher $(q-1)T_m$ pour obtenir le temps de réponse d'une instance du message τ_m ;
- C_m correspond au *temps de propagation* du message sur le réseau.

Il est à noter qu'étant en non préemptif, une fois que le message commence à être transmis, cette transmission ne peut pas être interrompue. Ceci explique que lors du calcul du temps de réponse d'une instance q donnée, c'est un délai de mise en attente qui est pris en compte et non une interférence au sens propre. Le calcul de ce délai se fait en fonction du temps nécessaire aux messages plus prioritaires et aux instances précédentes du message étudié à être transmis. C'est également pour cette raison qu'une fois le délai d'attente évalué, il suffit de rajouter le temps de propagation pour obtenir le temps de réponse de l'instance.
De plus, si un message de priorité inférieure avait commencé sa transmission lors de l'arrivée du message τ_m, alors il est nécessaire d'attendre que ce message soit transmis intégralement.

Cette durée s'ajoute donc au délai que peux subir le message τ_m. Au final, nous avons donc [TB94] :

$$\omega_m(q) = \max_{k \in lp(m)} (C_k) + (q-1)C_m + \sum_{k \in hp(m)} \left\lceil \frac{\omega_m(q) + J_k + \tau_{bit}}{T_k} \right\rceil C_k \tag{2.23}$$

Ainsi, dans le cas d'un message, le système d'équations retenu pour déterminer son pire temps de réponse est le suivant :

$$\begin{cases} \omega_m(q) = \max_{k \in lp(m)} (C_k) + (q-1)C_m + \sum_{k \in hp(m)} \left\lceil \frac{\omega_m(q) + J_k + \tau_{bit}}{T_k} \right\rceil C_k \\ Tr_m = \max_{q=1,\ldots,Q} (\omega_m(q) + J_i - (q-1)T_i + C_m) \end{cases} \tag{2.24}$$

Dernier changement dû au caractère non préemptif, la détermination de Q. Il pouvait se faire "à la volée" dans le cas préemptif, grâce à la condition $Tr_i < T_i$. Malheureusement, ceci n'est plus valable dans le cas non préemptif [BLDB06]. Il est donc nécessaire de déterminer la longueur de la période d'activité L_m de niveau m, puis connaissant L_m, déterminer le nombre d'instances pouvant survenir, c'est-à-dire Q [DBBL07].
La détermination de la longueur L_m de la période d'activité m peut se faire grâce à la formule récursive suivante :

$$\begin{cases} L_m^{(0)} = C_m \\ L_m^{(j+1)} = \max_{k \in lp(m)} (C_k) + \sum_{k \in hep(m)} \left\lceil \frac{L_m^{(j)} + J_k}{T_k} \right\rceil C_k \end{cases} \tag{2.25}$$

où $hep(m)$ désigne l'ensemble des messages de priorité supérieure ou égale à celle de τ_m.
La connaissance de L_m permet alors de calculer facilement Q :

$$Q = \left\lceil \frac{L_m + J_m}{T_m} \right\rceil \tag{2.26}$$

VI.5. Pessimisme

L'analyse holistique n'est pas une méthode exacte. Elle introduit du pessimisme, comme l'illustre l'exemple suivant, tiré des travaux de Burns [BAW95]. Supposons que nous ayons deux tâches, τ_i et τ_j sur un même processeur, chacune en attente d'un message. Supposons également que τ_i est plus prioritaire que τ_j. Lors de la détermination du temps de réponse de τ_j en utilisant l'analyse holistique, nous prenons en compte l'interférence due à la tâche plus prioritaire τ_i, et ceci dans tous les cas. Or, comme le montre la figure 2.4, pendant la réception du message à destination de τ_j, la tâche τ_i peut commencer son exécution, et dans l'exemple donné, la terminer, évitant ainsi d'interférer avec l'exécution de la tâche τ_j. Nous voyons donc bien, ici, une mise en évidence du pessimisme introduit par l'analyse holistique.

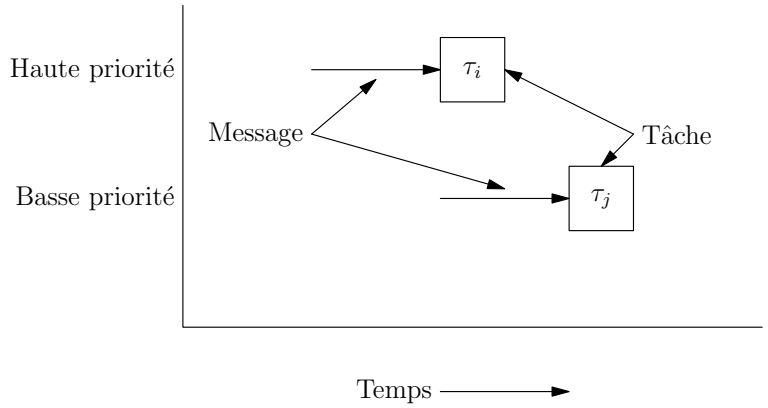


FIGURE 2.4 – Illustration du pessimisme de l'analyse holistique

Deuxième partie

Contributions

Chapitre 3

Algorithme d'optimisation du nombre de processeurs dans une architecture distribuée

Sommaire

Résumé

Ce chapitre présente une méthode de placement et d'ordonnancement de tâches sur des architectures distribuées tout en minimisant le nombre de processeurs utilisés.

I. Introduction

Grâce à leur haut potentiel en performance et fiabilité, les systèmes distribués sont utilisés dans un nombre croissant d'applications. Composés d'une architecture matérielle constituée de processeurs et de réseaux, sur laquelle vient s'exécuter des tâches, les systèmes temps réel ne cessent de gagner en complexité, comprenant toujours plus de processeurs et toujours plus de réseaux.
Mais cet accroissement de la taille des systèmes distribués n'est pas sans poser des questions de coûts, que ce soit des *coûts énergiques* (consommation électrique), des *coûts en poids* (par exemple, pour des systèmes embarqués à bord de drones ou de satellites), ou des *coûts pécuniers*.
Afin de diminuer ces coûts, nous avons développé un algorithme de minimisation du nombre de processeurs. Diminuer le nombre de processeurs permet de bien répondre à la problématique, car cela permet non seulement de diminuer les coûts directement liés au nombre de processeurs, mais aussi, permet de réduire la taille d'autres composants, notamment les batteries, au travers de la diminution du coût énergétique, et donc un gain de poids non négligeable.

La description de cet algorithme fait l'objet du courant chapitre, dans lequel nous ferons les hypothèses suivantes quant aux systèmes de tâches :
– aucune migration de tâches n'est autorisée ;
– chaque tâche et chaque message est affecté d'une priorité fixe ;
– les messages sont modélisés comme des tâches non préemptibles à priorité fixe (cf. CAN) ;
Afin de pouvoir présenter notre méthode de placement et d'ordonnancement de tâches tout en minimisant le nombre de processeurs utilisés, nous avons besoin d'introduire les travaux de M. Richard, réalisés dans le cadre de sa thèse [Ric02]. En effet, il a développé un algorithme de placement et d'ordonnancement pour des architectures distribuées qui est la base de notre algorithme d'optimisation.
Nous allons également présenter de manière plus précise l'algorithme FBB-FFD rapidement évoqué en section IV.4.b du chapitre 1, puisqu'il sera utilisé pour des comparaisons de performances.

II. FBB-FFD

Cet algorithme résulte du travail de Fisher, Baruah et Baker dans [FBB06], et permet de placer et d'ordonnancer des systèmes de tâches indépendantes et à départ simultané sur des systèmes multiprocesseurs.
Le principe est le suivant : considérons m processeurs et une liste de tâches $(\tau_1, \ldots, \tau_n)$ ordonnées par échéance croissante. Supposons que les tâches $\tau_1, \ldots, \tau_{i-1}$ ont été assignées à un processeur, et notons $\tau(P_\ell)$ l'ensemble des tâches assignées au processeur $P_\ell, 1 \leq \ell \leq m$.
Considérons maintenant que l'on veuille placer la tâche τ_i. Pour cela, nous allons affecter τ_i au premier processeur $P_k, 1 \leq k \leq m$, qui vérifie les relations suivantes (qui sont des conditions

suffisantes d'ordonnançabilité) :

$$\begin{cases} D_i - \sum_{\tau_j \in \tau(P_k)} C_j + \frac{C_j}{T_j} D_i \geq C_i \\ 1 - \sum_{\tau_j \in \tau(P_k)} \frac{C_j}{T_j} \geq \frac{C_i}{T_i} \end{cases} \tag{3.1}$$

S'il n'existe pas de processeur P_k permettant d'accueillir la tâche τ_i, alors l'algorithme ne peut ordonnancer le système de tâches.
Si toutes les tâches ont été placées, alors le système est ordonnançable en utilisant Deadline Monotonic (cf. chapitre 1 section III.1.b) comme politique d'ordonnancement sur chaque processeur.

III. Algorithme de placement et d'ordonnancement conjoints

L'originalité de l'algorithme développé par Michaël Richard lors de sa thèse [Ric02] repose sur *le placement et l'ordonnancement conjoints des tâches*, contrairement à la grande majorité des méthodes qui, dans un premier temps, placent les tâches sur les processeurs, puis dans un deuxième temps, les ordonnent.
En plus de cette originalité, la méthode développée est optimale vis-à-vis de l'analyse holistique. Autrement dit, s'il existe une solution qui peut être validée par l'analyse holistique, alors l'algorithme la trouvera.
Le principe de l'algorithme est de tester l'ensemble de toutes les combinaisons possibles. Si une solution est trouvée, alors le système est ordonnançable. Dans le cas contraire, le système n'est pas ordonnançable selon l'analyse holistique, ce qui signifie pas qu'il n'existe pas un ordonnancement valide car, rappelons-le, l'analyse holistique est une méthode qui introduit un certain pessimisme.
Dans la suite de ce document, l'algorithme développé par M. Richard et que nous nommons l'*algorithme de placement et d'ordonnancement conjoints* sera désigné sous le terme *APOC*.

III.1. Principe de la recherche

Comme dit plus haut, le principe est de tester l'ensemble de toutes les combinaisons possibles. Pour énumérer l'ensemble des combinaisons possibles, M. Richard se base sur l'exploration d'*un arbre de recherche*.
Si cette méthode est envisageable pour de petits systèmes, cela est inimaginable, d'un point de vue temps de calcul, pour des systèmes plus importants. Dans le but d'optimiser le temps de calcul, M. Richard a donc établi un certain nombre de règles, que nous pouvons classer en deux catégories :

– *règles de construction* : ces règles permettent de s'assurer que les solutions présentes dans l'arbre de recherche ne sont pas redondantes ;
– *règles d'élimination* : il s'agit d'éliminer au plus tôt les branches qui ne contiennent aucune solution valide.

Les règles précédentes permettent de diminuer les portions de l'arbre à explorer. Ensuite, pendant l'exploration, M. Richard a développé une méthode pour déterminer une borne inférieure du pire temps de réponse des tâches et des messages. Ainsi, si pendant le parcours de l'arbre, une branche mène à un système dont au moins une tâche (ou message) ne respecte pas son échéance, alors cette branche n'est pas explorée. En effet, si cette tâche (ou message) ne respecte pas son échéance avec une borne inférieure de son temps de réponse, alors elle ne le respectera pas avec son temps de réponse.

III.2. Structure de l'arbre de recherche

Pour décrire la structure de l'arbre, nous avons besoin d'introduire la notion de pool :

Définition 14. *un pool de processeurs est un ensemble de m processeurs identiques.*

Les messages étant modélisés comme des tâches, chaque réseau est assimilé à un pool de processeurs contenant un seul processeur, dont la politique d'ordonnancement est non préemptive. L'énumération de l'ensemble des permutations des tâches sur les différents processeurs se fait pool de processeurs par pool de processeurs, puis vient alors le ou les pool(s) réseau.
Il est nécessaire de respecter cet ordre d'énumération des pools, puisque l'ensemble des messages qui transiteront par un réseau ne sera connu qu'une fois toutes les tâches placées sur un processeur.

III.2.a. Arborescence d'un pool

La difficulté de la structure arborescente d'un pool consiste à énumérer l'ensemble des permutations possibles, pour un nombre de processeurs donné. La structure arborescente est constituée de *nœuds*, chaque nœud représentant une tâche.

Pour énumérer toutes les permutations au sein de l'arbre, deux types de noeuds sont utilisés (cf. figure 3.1 et [BFR75]) :
– si un chemin passe par un *nœud rond*, alors la tâche symbolisée par ce nœud est affectée au processeur courant ;
– si un chemin passe par un *nœud carré*, alors la tâche symbolisée par ce nœud est affectée au processeur suivant, qui devient le processeur courant.

De plus, les priorités des tâches sont affectées dans l'ordre décroissant lors du parcours d'un chemin. Ainsi, une tâche sur un noeud carré aura toujours la priorité la plus forte sur son processeur.

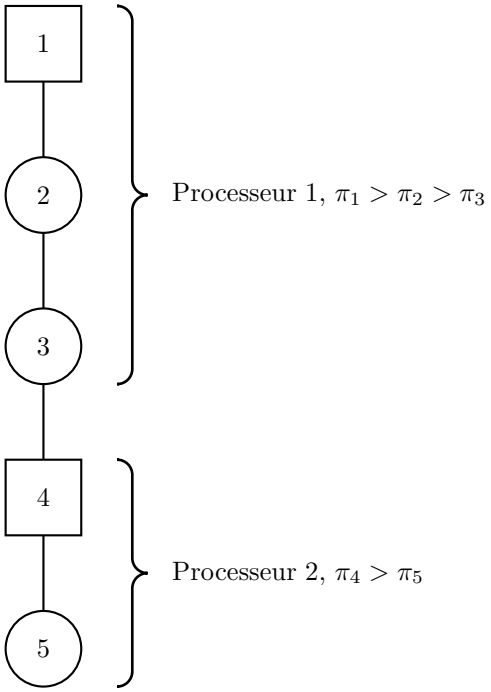


FIGURE 3.1 – Nœuds ronds et carrés

III.2.b. Phénomène de redondance des solutions

L'énumération telle qu'elle est présentée conduit à un phénomène de *redondance*, qui est représenté figure 3.2. En effet, certaines des branches sont équivalentes. C'est, par exemple, le cas des branches b) et q), qui modélisent chacune un processeur sur lequel se trouve la tâche τ_3 et un processeur sur lequel se trouvent les tâches τ_1 et τ_2.
Etant sur un pool de processeurs, tous les processeurs sont identiques, les deux branches sont donc équivalentes, d'où le phénomène de redondance.

III.3. Règles de construction de l'arbre de recherche

Les règles décrites ci-dessous ont pour but d'éviter tout phénomène de redondance lors du parcours de l'arbre (cf. figures 3.2 et 3.3) et de couper au plus tôt les branches ne conduisant pas au respect des contraintes du systèmes, comme les contraintes de précédence :

1. Le niveau 0 de la structure arborescente contient un unique nœud fictif représentant la racine de l'arbre ;

2. Le niveau 1 est formé de $n - m + 1$ nœuds carrés. Ceci correspond au placement des $n - m + 1$ premières tâches sur le premier processeur. Ces tâches seront ordonnancées au niveau de priorité le plus fort.
3. un chemin partant du niveau 0 et se terminant au niveau $i, 1 \leq i \leq n$, peut être agrandi au niveau $i + 1$ par n'importe quel nœud rond parmi n ou n'importe quel nœud carré parmi n, si les règles 4, 5 et 6 sont respectées.
4. le numéro k d'une tâche τ_k n'apparaît qu'une seule fois dans tout le chemin de la racine à une feuille de l'arbre. Ceci assure qu'une tâche ne sera placée qu'une seule fois.
5. Un nœud carré k ne peut-être utilisé pour agrandir un chemin contenant déjà un nœud carré l tel que $l > k$. Ceci évite les redondances d'énumération, c'est-à-dire la construction de chemins différents mais modélisant le même placement des tâches et la même affectation des priorités.
6. Aucun chemin ne peut être étendu par un nœud carré quelconque si ce chemin contient déjà m nœuds carrés. Cette règle assure le respect du nombre de processeurs.
7. Aucun chemin ne se termine s'il contient moins de m nœuds carrés, sauf si $n < m$. Ceci assure que tous les processeurs du pool sont utilisés.
8. Soient deux tâches en précédence, $\tau_k \prec \tau_l$; si τ_k et τ_l sont placées sur le même processeur, alors il n'existe pas de sous chemin débutant par un nœud carré l ou un nœud rond l et se terminant par un nœud rond k ne contenant que des nœuds entre l'origine et la fin du sous chemin. Ceci assure que la relation de précédence est vérifiée pour les tâches τ_k et τ_l
9. Un sous-chemin débutant par un nœud carré k et composé de l nœuds ne peut être étendu par un nouveau nœud rond r si la charge des tâches composant le sous-chemin, notée U_s, additionnée à la charge engendrée par τ_r est strictement supérieure à 1. Si $U_s + U_{\tau_r} > 1$ et $r > k$, alors un nœud carré r sera créé.
10. Soit τ'_k la tâche à insérer. On note $nb_{\tau_{k'}}$ le nombre de tâches τ_r non encore placées et telles que $r \geq k'$. Soit nb_P le nombre de processeurs non utilisés : si $nb_\tau < nb_P$, alors le nœud carré k' n'est pas créé.
11. Soit τ'_k la tâche insérée dans le dernier nœud carré et $\tau_{k''}$ la tâche à insérer. Si $nb_{\tau_{k'}} < nb_P$, alors le nœud rond k" n'est pas créé.

III.4. Evaluation du pire temps de réponse des tâches et des messages

Le pire temps de réponse d'une tâche dépend principalement de deux choses. Tout d'abord, de l'*interférence* due aux tâches plus prioritaires placées sur le même processeur, ensuite du *temps de réponse de ses prédécesseurs*, notamment à travers les gigues sur activation.
Le problème qui se pose est que les gigues dépendent du temps de réponse des tâches, qui dépend des gigues. Pour pouvoir calculer le temps de réponse des tâches, on procède donc de la manière suivante :

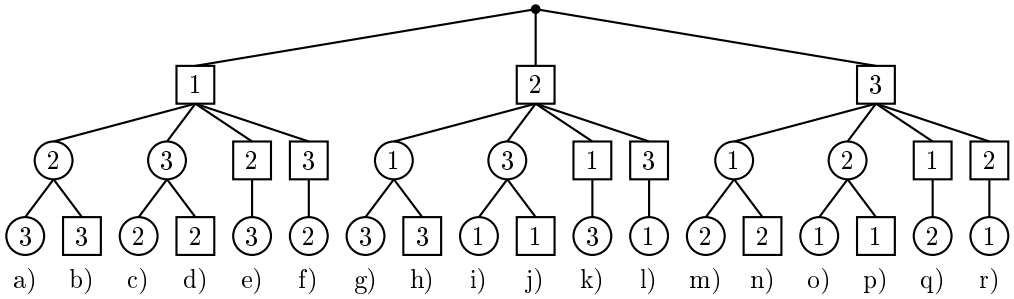


FIGURE 3.2 – Redondances lors de l'énumération

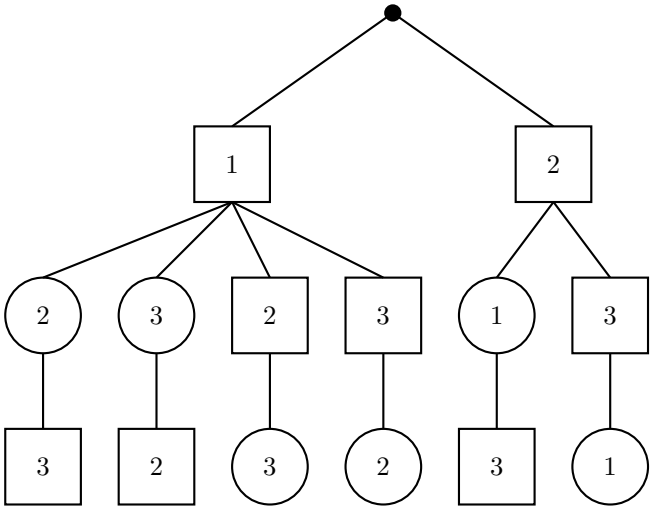


FIGURE 3.3 – Arbre obtenu en respectant les règles de construction

1. on initialise les gigues à zéro ;
2. on calcul les temps de réponse, sans mettre à jour les gigues ;
3. on calcul les gigues, sans mettre à jours les temps de réponses ;
4. si au moins une tâche à un temps de réponse différent entre l'itération courante et la précédente, alors on retourne à l'étape 2.

Il s'agit du même principe que celui utilisé lors du calcul des temps de réponse grâce à l'analyse holistique (cf. section VI du chapitre 2).

III.4.a. Notations

- Γ_i^{-P} : ensemble des messages émis par les prédécesseurs de τ_i, qui sont placés sur un processeur différent au moment de l'analyse de τ_i.
- Γ_i^{-NP} : ensemble des messages émis par les prédécesseurs de τ_i, qui ne sont pas placés, mais qui appartiennent au pool courant.
- Γ_i^{*-NP} : ensemble des messages émis par les prédécesseurs de τ_i, qui ne sont pas placés et qui n'appartiennent pas au pool courant.
- Γ_i^{-C} : ensemble des prédécesseurs de τ_i, qui sont situés sur le même processeur que τ_i.
- $LB(Tr_i)$: borne inférieure du pire temps de réponse de la tâche τ_i.
- $LB(J_i)$: borne inférieure de la gigue sur l'activation de la tâche τ_i.
- Θ_i : Tâches restant à placer sur le pool Pl_i.
- P_c : Processeur courant.

III.4.b. Pour une tâche

Afin de simplifier les équations, nous supposons que les échéances sont contraintes (ie. $D_i < T_i$) et donc la détermination du temps de réponse de τ_i ne nécessite de ne considérer que la première instance de la tâche τ_i. Malgré cette simplification, il est facile de lever cette contrainte. En effet, l'influence d'instances supplémentaires ne se fait ressentir qu'au niveau du calcul de l'interférence. Il est donc tout à fait possible de tenir compte de multiples instances en se basant sur la méthode utilisée lors de l'analyse holistique (cf. section V.1 du chapitre 2).

- *Cas d'une tâche placée*
  Si une tâche est placée, alors l'ensemble des tâches plus prioritaires est connu. Il est donc possible de calculer l'interférence due à ces tâches.

$$Int^{(k+1)} = C_i + \sum_{j=0}^{l-1} \left\lceil \frac{LB(J_j) + Int^{(k)}}{T_j} \right\rceil C_j \tag{3.2}$$

$$LB(Tr_i) = Int + LB(J_i) \tag{3.3}$$

- *Cas d'une tâche non placée*

– *P_c est le dernier processeur du pool*
Dans le cas où la tâche n'est pas placée, si le processeur courant est le dernier du pool, alors les tâches déjà placées sur ce processeur constituent un sous-ensemble des tâches plus prioritaires que la tâche τ_i. De plus, dans le cadre de cette thèse, nous avons constaté qu'il était également possible de prendre en compte dans cet ensemble de tâche plus prioritaire l'ensemble des prédécesseurs de τ_i appartenant à ce pool et qui ne sont pas encore placés. M. Richard ne prenait pas en compte ces tâches dans le cadre de ses travaux.

$$Int^{(k+1)} = C_i + \sum_{j=0}^{l} \left\lceil \frac{LB(J_j) + Int^{(k)}}{T_j} \right\rceil C_j \tag{3.4}$$

$$LB(Tr_i) = Int + LB(J_i) \tag{3.5}$$

– *P_c n'est pas le dernier processeur du pool*
Dans le cas présent, il n'est pas possible de connaître un sous-ensemble des tâches plus prioritaires.

$$LB(Tr_i) = LB(J_i) + C_i \tag{3.6}$$

III.4.c. Pour un message

– *Le message m_i est placé*
Si le message est placé, alors on connaît l'ensemble des messages plus prioritaires, et il est donc aisé de calculer l'interférence induite par ces messages. L'ensemble des messages moins prioritaires est constitué des messages affectés d'une priorité inférieure ainsi que de l'ensemble des messages restants à placer sur le pool réseau θ_r.

$$\begin{cases} \omega_m(q) = \max\limits_{k \in lp(m) \cup \theta_r} (C_k) + (q-1)C_m + \sum\limits_{k \in hp(m)} \left\lceil \dfrac{\omega_m(q) + J_k + \tau_{bit}}{T_k} \right\rceil C_k \\ LB(Tr_m) = \max\limits_{q=1,\ldots,Q} (\omega_m(q) + LB(J_i) - (q-1)T_i + C_m) \end{cases} \tag{3.7}$$

– *Le message m_i n'est pas placé*
  – *Le pool courant est le pool réseau*
  Si le pool courant est le pool réseau, alors toutes les tâches sont placées. Les messages déjà placés et ayant une priorité supérieure au message m_i constituent alors un sous-ensemble des messages de priorité plus élevée que le message courant. De même, ne connaissant pas l'ensemble des messages ayant une priorité inférieure au message m_i, nous ne considérerons que le sous-ensemble des messages déjà placés et ayant une priorité inférieure à m_i.

$$\begin{cases} \omega_m(q) = \max\limits_{k \in lp(m)} (C_k) + (q-1)C_m + \sum\limits_{k \in hp(m)} \left\lceil \dfrac{\omega_m(q) + J_k + \tau_{bit}}{T_k} \right\rceil C_k \\ LB(Tr_m) = \max\limits_{q=1,\ldots,Q} (\omega_m(q) + LB(J_i) - (q-1)T_i + C_m) \end{cases} \tag{3.8}$$

– *Le pool courant n'est pas le pool réseau*
Les messages n'étant pas tous placés, nous n'avons aucune information sur le réseau. Le temps de réponse s'obtient simplement par :

$$LB(Tr_i) = C_i + LB(J_i) \tag{3.9}$$

III.4.d. Evaluation de la gigue au démarrage pour une tâche

– *Pour une tâche sans prédécesseurs*

$$LB(J_i) = 0 \tag{3.10}$$

– *Tâches avec prédécesseurs*
  – *cas d'une tâche placée*
  Il n'est pas nécessaire de tenir compte des prédécesseurs déjà placés se trouvant sur le même processeur que la tâche τ_i. En effet, dans ce cas, l'influence que peut avoir le prédécesseur interviendra au niveau de l'interférence, et non au niveau de la gigue.

$$LB(J_i) = \max_{k \in \Gamma_i^{-P} \cup \Gamma_i^{-NP} \cup \Gamma_i^{*-NP}} (LB(Tr_k)) \tag{3.11}$$

  – *cas d'une tâche non placée*
    – $\tau_i \notin \Theta_c$ *: la tâche* τ_i *n'appartient pas au pool courant* Θ_c

$$LB(J_i) = \max_{k \in \Gamma_i^{-P} \cup \Gamma_i^{*-NP}} (LB(Tr_k)) \tag{3.12}$$

    – $\tau_i \in \Theta_c$ *: la tâche* τ_i *appartient au pool courant* Θ_c
      – P_c *est le dernier processeur du pool*

$$LB(J_i) = \max_{k \in \Gamma_i^{-P} \cup \Gamma_i^{*-NP}} (LB(Tr_k)) + \sum_{k \in \Gamma_i^{-C}} C_k \tag{3.13}$$

      – P_c *n'est pas le dernier processeur du pool*
      Dans ce cas, il faut envisager les deux placements possible, sur le processeur courant et sur un autre processeur, et, dans le but de rester optimal, prendre le plus petit temps de réponse entre les deux.

sur le processeur courant

$$\begin{array}{rcl} Int_i^{(k+1)} & = & \displaystyle\sum_{j=0}^{l} \left\lceil \frac{LB(J_j) + Int_i^{(k)}}{T_j} \right\rceil C_j \\ Int_i & = & Int_i^{(k+1)} = Int_i^{(k)} \\ LB_1(J_i) & = & Int_i + \displaystyle\max_{k \in \Gamma_i^{-P} \cup \Gamma_i^{*-NP}} (LB(Tr_k)) \end{array} \tag{3.14}$$

sur un autre processeur

$$LB_2(J_i) = \max_{k \in \Gamma_i^{-P} \cup \Gamma_i^{*-NP} \cup \Gamma_i^{-NP'}} (LB(Tr_k)) \tag{3.15}$$

D'où, finalement :

$$LB(J_i) = \min\left(LB_1(J_i), LB_2(J_i)\right) \tag{3.16}$$

III.4.e. Evaluation de la gigue au démarrage pour un message

La gigue sur l'activation dans le cas d'un message, qui dépend uniquement de la tâche émettrice τ_k, est donnée par l'équation suivante :

$$LB(J_i) = Tr_k \tag{3.17}$$

III.5. Règles de branchement

Les règles de branchement permettent de définir un ordre pour le parcours des pools et des tâches.

III.5.a. Ordre des pools de processeurs

On définit pour chaque pool un coefficient, dit *cœfficient de charge globale*, qui n'est autre que la somme des facteurs d'utilisation des tâches présentes sur le pool.

$$U_{Pl_i} = \sum_{k=1}^{n_i} \frac{C_k}{T_k} \tag{3.18}$$

Les pools sont alors ordonnés selon leur charge décroissante. Ainsi, le pool ayant la plus forte charge globale est situé au début de la structure arborescente. Etant le plus chargé, c'est lui qui possède le moins de solutions valides. Ceci permet de réaliser de nombreuses coupes au début de la structure arborescente, limitant ainsi le nombre de chemins à parcourir.

III.5.b. Ordre des tâches

De la même manière que pour les pools de processeurs, les tâches sont triées par facteur d'utilisation décroissant. Ainsi, lors d'un parcours en profondeur de l'arbre de recherche, les tâches avec un facteur d'utilisation important étant plus difficiles à ordonnancer, le nombre de coupes effectuées dans chaque sous-arbre est plus important, réduisant d'autant plus le nombre de solutions à explorer.

III.6. Règles de sélection

III.6.a. Au sein d'un pool de processeurs

Il existe deux types de parcours différents :
- *parcours de remplissage* : Les nœuds privilégiés sont les nœuds ronds. Les tâches sont ajoutées au processeur courant tant que cela est possible. Les premières solutions obtenues lors du parcours de l'arbre sont alors déséquilibrées, dans le sens où les derniers processeurs peuvent avoir un taux d'utilisation faible par rapport aux premiers processeurs.
- *parcours d'équilibrage* : Ajouter un critère de sélection permet d'éviter le phénomène précédent, en créant un nœud carré dès que le critère est vérifié. Pour un pool Pl_i contenant m_i processeurs et sur lequel n_i tâches doivent être placées, deux critères sont envisageables :
  - *fonction du nombre de tâches* Les nœuds ronds sont construits tant que le nombre de tâches sur le processeur courant est inférieur à :

$$\left\lceil \frac{n_i}{m_i} \right\rceil \tag{3.19}$$

  - *fonction du taux d'utilisation* : Les nœuds ronds sont favorisés tant que le taux d'utilisation du processeur courant est inférieur à :

$$\frac{U_{Pl_i}}{m_i} \tag{3.20}$$

III.6.b. Au sein de l'arbre global

Le parcours de l'arbre en profondeur privilégie les branches les plus à gauche de l'arbre. Dans le but de pouvoir explorer l'arbre de manière plus uniforme, plusieurs types de parcours ont été étudiés :
- *parcours en profondeur simple* : c'est le parcours classique, la branche la plus à gauche de l'arbre est parcourue en premier ;
- *parcours en parallèle* : plusieurs branches de l'arbre sont parcourues en même temps. Pour cela, l'arbre global est divisé en plusieurs parties. Plus précisément, l'arbre global est découpé au niveau de son premier niveau de profondeur, les $n_1 - m_1 + 1$ sous-arbres étant alors explorés en même temps.

- *parcours en parallèle auto-adaptatif* : Le nombre de branches contenues dans chaque sous-arbre n'étant pas égal, parcourir équitablement le sous-arbre de gauche et celui de droite n'est équitable que d'un point de vue temporel. Afin de pallier ce problème, un sous-arbre peut-être favorisé, c'est-à-dire être parcouru plus longtemps que les autres. Ici encore, deux politiques différentes se confrontent :
  - L'arbre contenant le moins de branches est privilégié. Si une solution existe, elle sera trouvée rapidement.
  - L'arbre contenant le plus de branches est privilégié. La probabilité de trouver une solution est plus élevée.

IV. Algorithme de minimisation du nombre de processeurs

L'*algorithme de minimisation du nombre de processeurs* (ou *AMNP*) a été développé dans le but de minimiser le nombre de processeurs nécessaires pour l'ordonnancement d'un système de tâches, selon l'analyse holistique. Cet algorithme reprend les principes de base de l'algorithme précédent (cf. section III), et y apporte des modifications, notamment, au niveau de l'arbre de recherche pour permettre d'ajouter un critère d'optimisation.
L'APOC prenait comme paramètres, entre autres, le système de tâches et le nombre de processeurs. L'AMNP s'affranchit du nombre de processeurs, et va déterminer le nombre de processeurs optimal.
Pour ce faire, on détermine d'abord le *nombre minimum* de processeurs nécessaires pour que le système étudié soit potentiellement ordonnançable (ie. en deçà de ce nombre, le système ne sera pas ordonnançable), et un *nombre maximum* de processeurs, au delà duquel une recherche est inutile, par exemple, en cas d'une solution déjà connue. Ceci permet d'encadrer le nombre de processeurs qui seraient nécessaires pour ordonnancer fiablement le système de tâches.
L'algorithme créé alors un arbre de solutions, basé sur des règles de l'APOC mais adaptées pour tenir compte non plus d'un nombre fixe de processeurs, mais d'un intervalle de processeurs. L'algorithme parcourt ensuite l'ensemble des solutions. Lorsqu'une solution est trouvée, la borne supérieure du nombre de processeurs est mise à jour, et la recherche continue dans le but d'essayer de trouver une solution meilleure (c'est-à-dire, qui utilise un nombre moindre de processeurs).
A noter également que cet algorithme, tout comme l'algorithme de placement et d'ordonnancement conjoints, est optimal vis-à-vis de l'analyse holistique.

IV.1. Méthodes de construction de l'arbre de recherche

Deux méthodes sont envisageables pour la construction de l'arbre de recherche.
- soit on construit un arbre dont le premier sous-arbre contient l'ensemble des solutions utilisant le nombre minimum de processeurs, le deuxième sous-arbre contenant l'ensemble des solutions utilisant le nombre minimum de processeurs augmenté de 1, etc... jusqu'au dernier sous-arbre

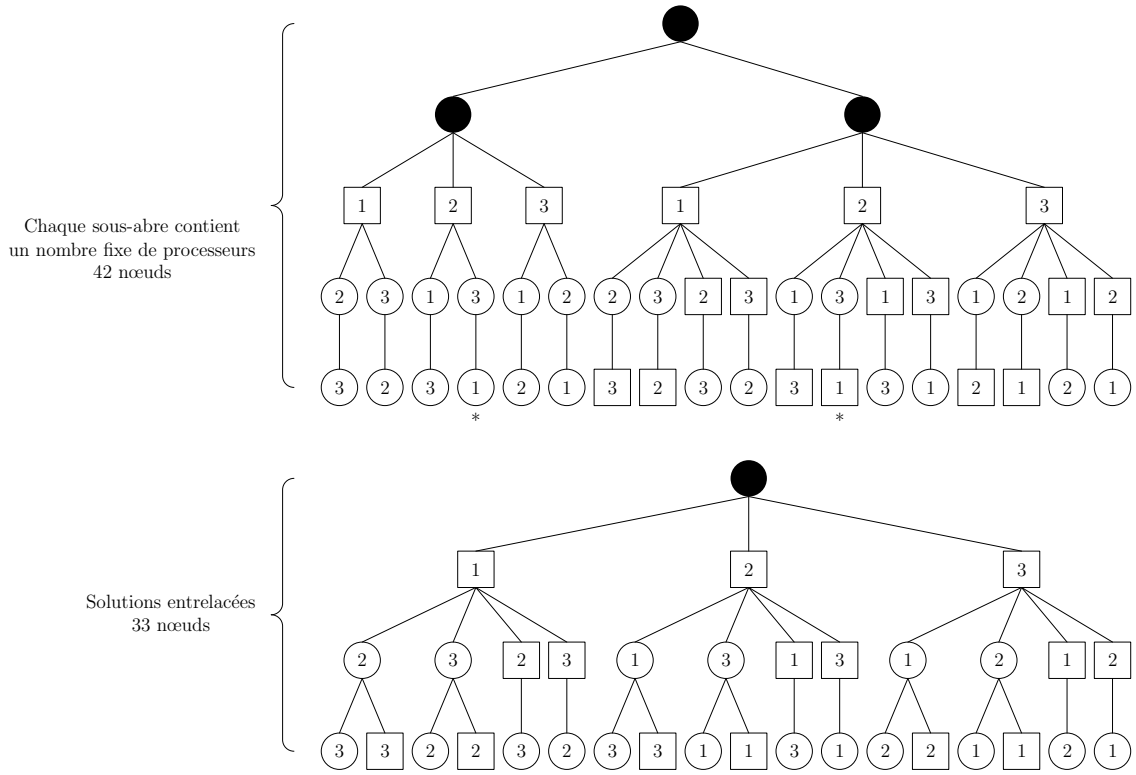


FIGURE 3.4 – Différents arbres d'exploration possibles

qui contient les solutions utilisant le nombre maximum de processeurs. Cette approche revient à utiliser l'APOC en faisant varier le nombre de processeurs.
– soit on construit un arbre de recherche ne privilégiant pas le nombre de processeurs, et qui contient donc toutes les solutions entrelacées.

Ces deux méthodes sont illustrées figure 3.4. Pour des raisons de clarté, aucune règle de construction n'a été appliquée, d'où la présence de redondances. Les sous-arbres sont construits pour un nombre minimum de processeurs égal à un et un nombre maximum égal à deux.

La méthode retenue est la deuxième, pour plusieurs raisons. Outre le fait que la première méthode consiste à appeler l'APOC plusieurs fois de suite en faisant uniquement varier le nombre de processeurs, la deuxième solution conduit à un arbre ayant moins de nœuds, donc son exploration nécessite moins de calculs. Cette diminution du nombre de nœuds est due à la factorisation des parties communes aux branches : comme le montre la figure 3.4, les deux chemins de l'arbre supérieur marqués par des * ne diffèrent que par la dernière tâche. L'arbre inférieur est construit de sorte qu'il factorise les parties communes entre les branches.

Dans la suite de ce chapitre, nous allons donc développer la seconde méthode.

IV.2. Nombre de processeurs

Les bornes inférieure et supérieure du nombre de processeurs sont susceptibles d'évoluer au cours de l'exploration de l'arbre des solutions. Les valeurs données ci-dessous sont les valeurs utilisées pour initialiser les bornes, avant de commencer la recherche.

IV.2.a. Borne inférieure

Comme dit un peu plus haut, le but de la borne inférieure est de pouvoir dire qu'en dessous de ce nombre, on sait que le système ne sera pas ordonnançable et reflète donc une condition nécessaire d'ordonnancement.
La borne inférieure actuellement utilisée se base sur la charge totale du système de tâches à placer et à ordonnancer. En effet, une condition nécessaire pour qu'un système de tâches soit ordonnançable est que le nombre de processeurs disponibles soit supérieur ou égal à la charge totale des tâches.

$$min = \left\lceil \sum_i \frac{C_i}{T_i} \right\rceil \tag{3.21}$$

IV.2.b. Borne supérieure

La borne supérieure du nombre de processeurs est un peu plus difficile à définir. La borne inférieure étant une condition nécessaire, ou pourrait être tenté de définir la borne supérieure comme une condition suffisante. Malheureusement, il existe des systèmes de tâches non ordonnançable sur n processeurs, mais qui le seront sur $n-1$ processeurs, comme le montre l'exemple suivant.
Soient deux tâches τ_1 et τ_2, avec une même période T_i de 5, une même échéance relative D_i de 2 et une durée d'exécution C_i de 1. Supposons de plus que τ_1 envoi un message τ_m à τ_2.
Si on place τ_1 et τ_2 sur deux processeurs différents, alors nous voyons que la tâche τ_2 manque inévitablement son échéance, ceci à cause du délai de propagation du message τ_m. Par contre, si les deux tâches sont sur le même processeur, alors le délai de propagation du message est nul, et τ_2 respecte alors son échéance.
Malgré cette impossibilité de définir la borne supérieure comme étant une condition suffisante, il est possible d'initialiser cette borne supérieure avec le nombre de tâches composant le système. Dans ce cas, un scénario possible est d'affecter à chaque processeur une et une seule tâche. La migration des tâches n'étant pas autorisée, l'ajout d'un processeur supplémentaire serait ici inutile, puisqu'il resterait tout le temps oisif. Nous avons donc initialisé le nombre maximum de processeurs à utiliser par :

$$max = \text{nombre de tâches du système} \tag{3.22}$$

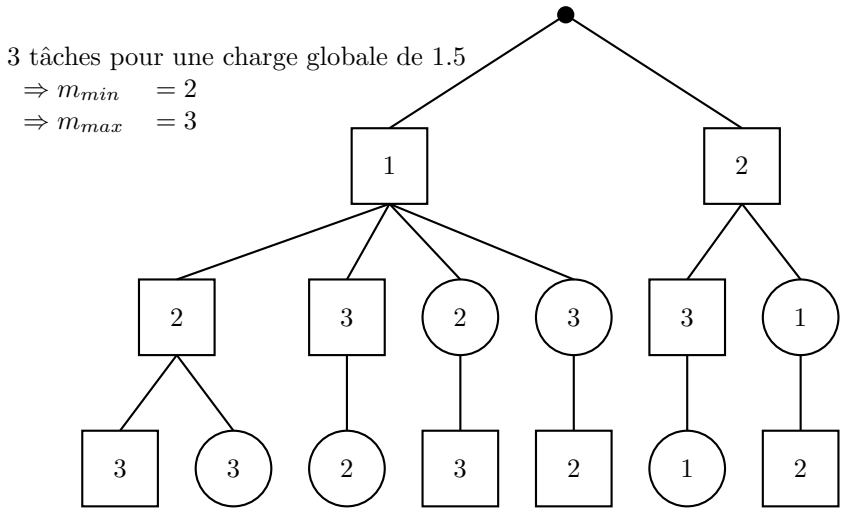


FIGURE 3.5 – Arbre des solutions

IV.3. Règles

IV.3.a. Synthèse

Les *règles en italiques* correspondent aux règles adaptées des travaux de Richard [Ric02] ou créées.

Règles de construction L'arbre de recherche correspondant à un pool composé de n tâches, est défini par les règles suivantes, qui permettent d'éviter les redondances lors de l'énumération des solutions :

- Règle n°1 : Le niveau 0 de la structure arborescente contient un unique nœud fictif représentant la racine de l'arbre.
- *Règle n°2* : Le niveau 1 est formé de $n - m_{min} + 1$ nœuds carrés.
- Règle n°3 : Un chemin partant du niveau 0 et se terminant au niveau $i, 1 \leq i \leq n$, peut être agrandi au niveau $i+1$ par n'importe quel nœud rond parmi n ou n'importe quel nœud carré parmi n, si les règles 4, 5 et 6 sont respectées.
- Règle n°4 : Le numéro k d'une tâche τ_k n'apparaît qu'une seule fois dans tout le chemin de la racine à une feuille de l'arbre. Ceci assure qu'une tâche ne sera placée qu'une seule fois.
- Règle n°5 : U nœud carré k ne peut être utilisé pour agrandir un chemin contenant déjà un nœud carré l tel que $l > k$. Ceci évite les redondances d'énumération, c'est-à-dire la construction de chemins différents mais modélisant le même placement des tâches et la même affectation des priorités.

- *Règle n°6* : Aucun chemin ne peut être étendu par un nœud carré quelconque si ce chemin contient déjà m_{max} nœuds carrés.
- *Règle n°7* : Aucun chemin ne se termine s'il contient moins de m_{min} nœuds carrés.

Règles de coupe Les règles de coupe permettent de s'assurer que les contraintes du système sont respectées, comme les contraintes liées aux relations de précédence ou encore le taux d'utilisation de chaque processeur :

- Règle n°8 : Soient deux tâches en précédence, $\tau_k \prec \tau_l$; si τ_k et τ_l sont placées sur le même processeur, alors il n'existe pas de sous chemin débutant par un nœud carré l ou un nœud rond l et se terminant par un nœud rond k ne contenant que des nœuds ronds entre l'origine et la fin du sous chemin. Ceci assure que la relation de précédence est vérifiée pour les tâches τ_k et τ_l.
- Règle n°9 : Un sous-chemin débutant par un nœud carré k et composé de l nœuds ronds ne peut être étendu par un nouveau nœud rond r si la charge des tâches composant le sous-chemin, notée U_s, additionnée à la charge engendrée par la tâche τ_r est strictement supérieure à 1. Si $U_s + U_r > 1$ et $r > k$, alors un nœud carré r sera créé.
- Règle n°10 : Soit τ_k la tâche à insérer. On note nb_{τ_k} le nombre de tâches τ_r non encore placées telles que $r \geq k'$. Soit nb_{Pr} le nombre de processeurs non utilisés : si $nb_k > nb_{Pr}$ alors le nœud carré k n'est pas créé.
- *Règle n°11* : Soit U la charge des tâches non encore placées et soit nb_{Pr} le nombre de processeurs non utilisés. Si $U > nb_{Pr}$ alors le nœud carré n'est pas créé. Cela évite de construire les chemins qui conduiront obligatoirement un processeur ou plus à avoir une charge strictement supérieure à 1.
- *Règle n°12* : Soit τ_k la tâche à insérer. Si le processeur courant est le dernier, et s'il existe au moins un prédécesseur de τ_k non encore placé, alors le nœud rond k n'est pas créé. Ceci évite de construire des branches qui conduiront inévitablement à une violation de la règle n°8.

Règles de performance Cet ensemble de règle est spécifique à notre méthode permettant de minimiser le nombre de processeurs utilisés. En effet, avec les règles de performance, nous nous assurons qu'aucune solution utilisant un nombre de processeurs supérieur au nombre de processeurs utilisés par la dernière solution valide ne sera énumérée :

- *Règle n°13* : Lorsqu'une solution est trouvée, alors elle est stockée (elle remplace la précédente, si une solution avait déjà été trouvée), et le nombre de processeurs maximum m_{max} prend alors comme valeur le nombre de processeurs utilisés par la solution trouvée, moins 1.
- *Règle n°14* : Lors du parcours du niveau 1, soit i le numéro de la tâche en cours de traitement. Le nombre maximum de processeurs est :
  - inchangé si $i \leq n - m_{max} + 1$
  - égal au minimum de m_{max} et de $m_{max_0} - i + 1$ sinon.

  où m_{max_0} est le nombre maximum de processeurs au début de l'algorithme.

IV.3.b. Détails des règles

Règles adaptées De nombreuses règles ne sont qu'une adaptation des règles de l'APOC en vue de prendre en compte un intervalle de processeurs au lieu d'un nombre de processeurs fixés. Cela inclut les règles 1 à 10.

Règles d'optimisation du calcul Les règles 11 et 12 permettent d'éviter de construire des branches qui ne mèneront à aucun chemin valide.
En effet, dans le cas de la règle 11, si la charge des tâches restant à placer est supérieure au nombre de processeurs restants, alors aucun nœud carré n'est créé, puisque cela impliquerait tôt ou tard la présence d'au moins un processeur avec une charge supérieure à 1.
La règle 12 intervient lorsqu'on est sur le dernier processeur, et permet d'éviter de créer des chemins menant obligatoirement à une violation de contrainte de précédence (c'est-à-dire, avoir une tâche réceptrice possédant une priorité plus forte qu'une tâche émettrice).

Règles de performance Les règles de performance permettent de modifier le comportement de la recherche pendant l'exploration de l'arbre. Par exemple, la règle 13 permet, dès qu'une solution est trouvée, d'éliminer toute branche qui conduirait à une solution équivalente, sinon pire, en terme de nombre de processeurs.
La règle 14 est en fait une évolution de la règle 1 de l'algorithme de M. Richard. En effet, la règle 1 indique le nombre de nœuds du niveau 1 de l'arbre de recherche en fonction du nombre de processeurs. Puisque maintenant nous travaillons avec un intervalle de processeurs, il était nécessaire de faire évoluer cette règle.
La règle 14 est schématisée par la figure 3.6 sur un exemple composé de 3 tâches, pour lequel $m_{min} = 1$ et $m_{max} = 2$.

IV.4. Evaluation de la borne inférieure pour le pire temps de réponse et pour la gigue

Le principe du calcul de la borne inférieure pour le pire temps de réponse et pour la gigue sur l'activation est exactement le même que pour l'APOC (cf. section III.4).

IV.5. Règles de sélection

IV.5.a. Au sein de l'arbre global

Les règles de sélection sont les mêmes que celles employées pour l'APOC. Dans [Ric02], M. Richard montre que le parcours en parallèle auto-adaptatif présente globalement de meilleures performances que les autres. Pour cette raison, ce parcours a été sélectionné pour la réalisation de l'AMNP.

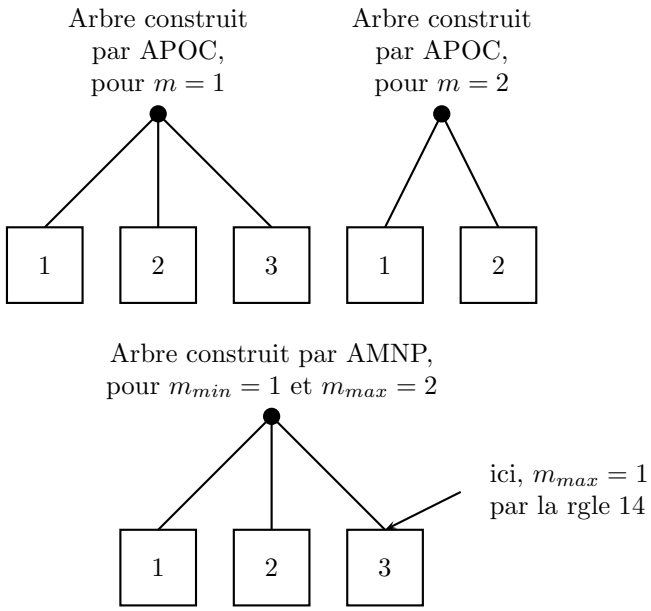


FIGURE 3.6 – Visualisation de la règle 14

IV.5.b. Au sein d'un pool de processeurs

Au sein d'un pool de processeurs, on peut distinguer deux comportements selon le type de tâches considérées :

- si les tâches sont indépendantes, alors une politique de remplissage des processeurs est parfaitement adaptée. En effet, les tâches étant indépendantes, l'allocation des tâches sur un processeur ne peux avoir d'influence sur le temps de réponse des tâches placées sur un autre processeur.
- si les tâches ne sont pas indépendantes, alors M. Richard a montré qu'une politique d'équilibrage conduisait à de meilleur résultat qu'une politique de remplissage (cf paragraphe III.6.a). Nous avons donc décidé d'utiliser une politique d'équilibrage, basée sur une charge moyenne l. Autrement dit, les branches qui seront privilégiées lors de l'exploration de l'arbre seront celles contenant des processeurs dont la charge est aux alentours de l. En effet, ne connaissant pas le nombre de processeurs de la solution optimale, il est impossible de déterminer une charge moyenne. Les tâches n'étant pas indépendantes, placer une tâche τ_i sur un processeur peut modifier le temps de réponse d'une tâche τ_j située sur un autre processeur et rendre ainsi la tâche τ_j non ordonnançable. Le choix de l est laissé à l'expérimentateur et est donc un paramètre sur lequel il est possible de jouer pour modifier le parcours de l'arbre

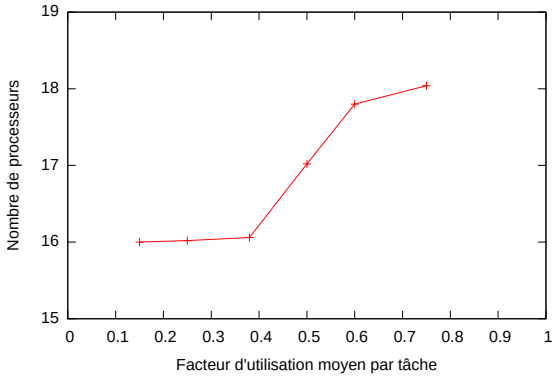


FIGURE 3.7 – Evolution du nombre de processeurs en fonction du nombre de tâches

de recherche. Une faible valeur de l favorisera l'algorithme à explorer des solutions ayant beaucoup de processeurs faiblement chargés, tandis qu'une forte valeur de l favorisera plutôt des solutions dont les processeurs sont moins nombreux et donc plus chargés. A noter que ce paramètre influence uniquement l'ordre dans lequel la recherche est effectuée, et donc le temps de calcul, et n'a aucune influence sur le résultat final.

IV.6. Résultats numériques

Afin d'étudier les performances de l'algorithme, des tests ont été réalisés. Lors de ces tests, sauf mention contraire, nous n'avons considéré que des systèmes de tâches indépendantes. Nous n'avons donc pas de réseaux et pouvons considérer que nos systèmes sont ordonnancés sur des architectures multiprocesseurs.

IV.6.a. Evolution des résultats en fonctions du nombre de tâches

Voici le récapitulatif de nos conditions expérimentales, qui nous ont permis d'obtenir les résultats de cette section :

- chaque configuration contient un nombre de tâches pris dans l'ensemble $\{20, 30, 40, 60, 100\}$;
- le facteur d'utilisation des tâches est généré en utilisant l'algorithme UUniFast, développé par Bini [BB04a], en vue de minimiser le biais introduit lors de la génération des systèmes de tâches ;
- les systèmes générés ont un facteur d'utilisation de 15 ;
- au bout de 5 minutes, si les calculs ne sont pas terminés, ils sont interrompus. La dernière solution trouvée par l'algorithme est alors renvoyée.

La figure 3.7 montre l'évolution du nombre de processeurs nécessaires pour ordonnancer un système de tâches en fonction du facteur d'utilisation moyen par tâche. Lorsque ce facteur d'utilisation moyen est faible, notre algorithme trouve une solution utilisant 16 processeurs

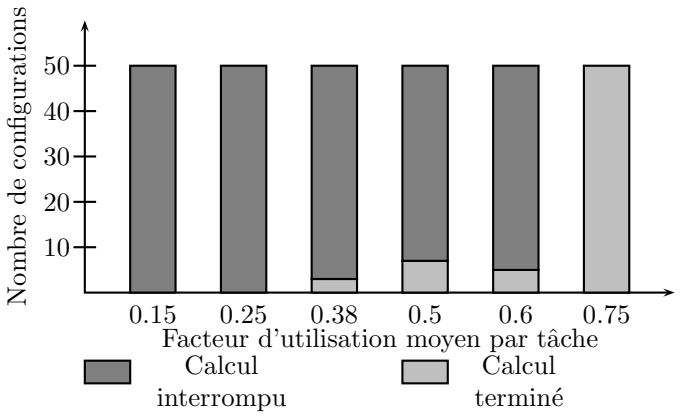


FIGURE 3.8 – Nombre de calculs terminés en fonction de la charge moyenne par tâche

pour un système de tâche nécessitant au minimum 15 processeurs puisque le facteur d'utilisation totale du système est de 15.

Lorsque le facteur d'utilisation moyen par tâche augmente, le nombre de processeurs augmente également puisque le système est plus contraint lorsque les tâches ont un facteur d'utilisation plus élevée. En effet, avec un ensemble de tâches ayant un facteur d'utilisation moyen élevé, nous avons moins de solutions à évaluer. Mais le pourcentage de solutions non ordonnançables est élevé comparé à un ensemble de tâches avec un facteur d'utilisation moyen faible par tâche. La figure 3.8 montre le nombre de calculs arrivés à terme pour chaque configuration en fonction du facteur d'utilisation moyen par tâche. Pour un facteur d'utilisation moyen faible, aucun calcul n'arrive à terme avant la fin du temps limite (fixé à 5 minutes). Tandis que pour les tâches avec un facteur d'utilisation moyen élevé, tous les calculs se terminent. De même que précédemment, nous expliquons ce résultat par le fait que plus le facteur d'utilisation moyen par tâche est élevé, plus le système est contraint, nous permettant alors d'éliminer de nombreuses branches très tôt dans l'arbre de recherche, minimisant ainsi le nombre de solutions à explorer.

En raison de sa complexité exponentielle, notre algorithme peut prendre énormément de temps pour terminer son exécution. Deux cas doivent alors être étudiés :

– Puisque nous stoppons la recherche au bout de 5 minutes, nous avons défini un critère pour évaluer la pertinence de la solution obtenue. Nous désignons par Δ une borne supérieure de la distance à l'optimum. En théorie, Δ est la différence entre le nombre de processeurs utilisés par la dernière solution trouvée par notre algorithme avant que nous ne l'interrompions et le nombre de processeurs d'une solution optimale. Mais en pratique, ne connaissant pas le nombre de processeurs d'une solution optimale, nous utilisons le nombre de processeurs pour lequel nous savons qu'en deçà, il n'y a pas de solutions, c'est-à-dire que nous utilisons la borne inférieure du nombre de processeurs. Plus formellement, si nous considérons un ensemble de n tâches dans un pool avec un facteur d'utilisation global U, alors le nombre de processeurs

utilisés est :

$$N_{opt} \stackrel{\text{def}}{=} \left\lceil \frac{U}{n} \right\rceil \tag{3.23}$$

En conséquence, Δ est une borne supérieure de la distance à l'optimum.

– Le processus de recherche s'est terminé normalement : dans ce cas, nous obtenons une solution optimale vis-à-vis de l'analyse holistique. Nous pouvons alors affirmer qu'il n'existe pas d'ordonnancement valide utilisant moins de processeurs que la solution retournée par l'algorithme. La valeur de Δ est alors 0.

Comme le montre la figure 3.9, Δ augmente avec le facteur d'utilisation moyen par tâche. Ceci s'explique par le fait que, comme montré à la figure 3.7, le nombre de processeurs croît avec le facteur d'utilisation moyen. La valeur nulle de Δ pour les très hauts facteurs d'utilisation moyens indique que dans ce cas précis, les calculs arrivent à terme et l'algorithme renvoie alors une solution optimale.

Enfin, la figure 3.10 montre le temps qui a été nécessaire pour trouver une solution. Une fois encore, 2 cas sont à distinguer :

– Si le processus de recherche est interrompu : le temps considéré est le temps qui a été nécessaire pour trouver la dernière solution valide ;
– Si le processus de recherche se termine normalement : le temps considéré est alors le temps nécessaire à l'exploration complète de l'arbre de recherche, c'est-à-dire le temps nécessaire pour trouver la dernière solution valide (comme précédemment), plus le temps nécessaire à l'énumération de toutes les solutions restantes afin de prouver l'optimalité de la solution retenue.

Comme nous pouvons le voir, notre algorithme fourni une solution en un temps relativement court, surtout si nous considérons le fait que le problème est $\mathcal{NP}$-Difficile au sens fort.

Toutes ces expérimentations nous montrent que nous pouvons trouver une solution valide utilisant un nombre de processeurs proche de l'optimal en un temps relativement court. Pour des systèmes de tâches ayant un faible facteur d'utilisation moyen par tâche, une solution utilisant 16 processeurs est rapidement trouvée. De plus, lorsque notre algorithme se termine normalement, la grande majorité des calculs effectués le sont pour prouver l'optimalité de la solution, c'est-à-dire pour évaluer l'ensemble des solutions restantes. Notons également que pour des facteurs d'utilisation moyens intermédiaires, le temps de calcul devient plus important. Dans ce contexte, l'arbre de recherche contient le plus possible de solutions à explorer et le temps de calcul alloué au test d'ordonnançabilité est également plus important.

IV.6.b. Evolution des résultats en fonctions du nombre de messages

Les temps de calculs varient en fonction du nombre de messages. La figure 3.11 et le tableau 3.1 montrent des statistiques réalisées sur 50 configurations de 100 tâches d'un facteur d'utilisation total de 10 pour 0, 5, 10, 15 et 20 messages, les calculs étant interrompus au bout de 2 heures. La solution optimale se trouve forcément entre la borne inférieure du nombre de processeurs et le nombre de processeurs de la dernière solution trouvée. Partant de ce constat, il est possible

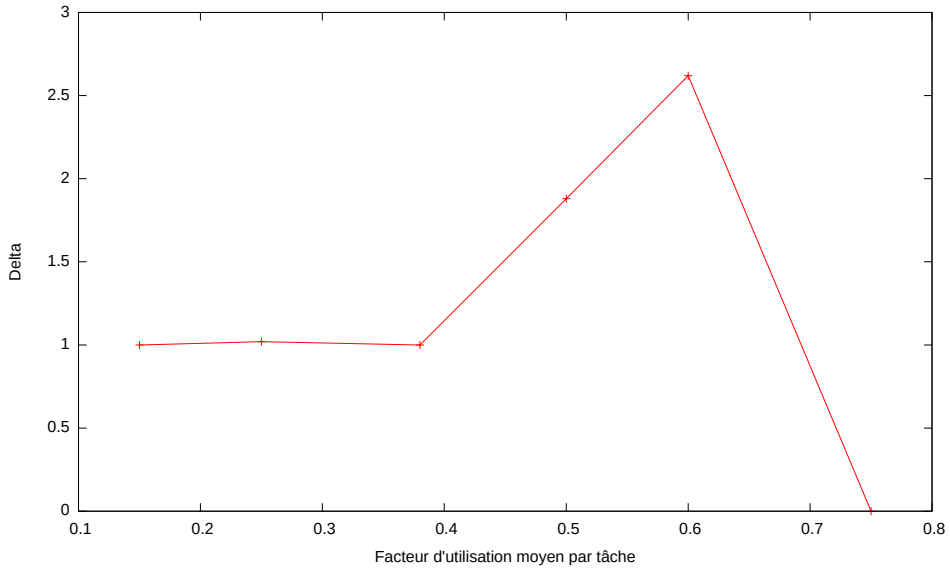


FIGURE 3.9 – Evolution de la distance à l'optimal

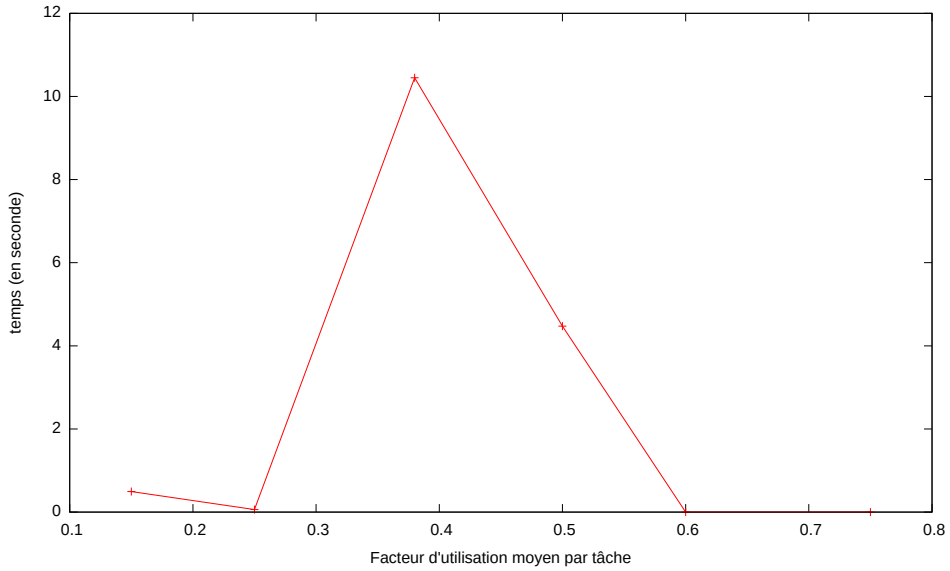


FIGURE 3.10 – Evolution du temps de calcul en fonction du facteur d'utilisation moyen par tâche

	0	5	10	15	20
t_{max} en s	1676,12	1232,74	674,82	4423,94	307,07
t_{moyen} en s	223,45	160,51	108,5	276,86	90,81

TABLE 3.1 – temps de calcul en fonction du nombre de messages

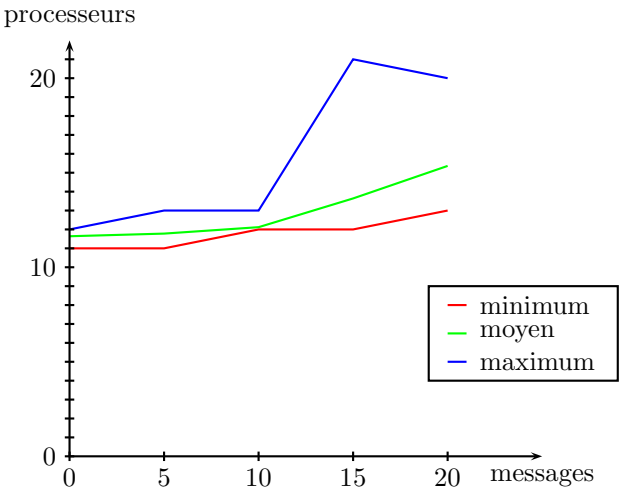


FIGURE 3.11 – Evolution du nombre de processeurs en fonction du nombre de messages

de dire :
- si le nombre de processeurs de la dernière solution trouvée est proche de la borne inférieure du nombre de processeurs, alors la solution trouvée est proche de la solution optimale ;
- si, au contraire, le nombre de processeurs de la dernière solution trouvée est éloignée de la borne inférieure du nombre de processeurs, alors il est impossible d'« estimer la distance » séparant le nombre de processeurs optimal du nombre de processeurs de la dernière solution trouvée.

Ainsi, d'après la figure 3.11, nous constatons que :
- pour un faible nombre de messages (≤ 10), l'intervalle dans lequel se situe la solution optimale est réduit. Il est donc possible d'affirmer que la solution trouvée à l'aide de l'algorithme est proche de la solution optimale ;
- pour un nombre de messages plus important (≥ 15), l'intervalle dans lequel se situe la solution optimale est large. Il n'est donc pas possible de garantir la qualité de la solution.

Il est à noter que les constatations précédentes ont été faites sur des résultats statistiques. C'est-à-dire qu'il s'agit d'une tendance générale. Le seul moyen de savoir si l'algorithme donnera une solution sur laquelle il sera possible de faire des pronostics ou non pour un système de tâches particulier est d'exécuter l'algorithme sur ce même système.

IV.6.c. Influence du temps de calcul

En reprenant les configurations précédentes, et en interrompant cette fois-ci le calcul au bout de 2 heures, nous constatons que le nombre de processeurs de la solution est inchangé.

Cela signifie que l'AMNP trouve rapidement une solution acceptable, et qu'il peine à prouver l'optimalité de la solution courante lorsque l'arbre restant à explorer contient peu ou pas de solutions.
Ainsi, il est possible d'arrêter le calcul au bout d'un temps qui est laissé à l'appréciation de l'expérimentateur, selon la qualité de la solution souhaitée. Toutefois, un temps relativement court peut fournir un résultat très acceptable, proche de l'optimal, dans le cas de tâches indépendantes.

IV.6.d. Temps d'obtention d'une solution optimale

Pour obtenir une solution optimale, il suffit de ne pas interrompre le calcul, et de le laisser aboutir.
Afin d'essayer d'évaluer le temps nécessaire à la résolution d'un système complexe (composé de 100 tâches), nous avons lancé 5 calculs. Au bout de 3 semaines, sur les 5 calculs lancés, aucun n'était terminé, et malheureusement, le calcul était loin d'être achevé. C'est pourquoi nous ne disposons pas de solution optimale, nécessaire à la poursuite des comparaisons, ni même une estimation du temps nécessaire pour obtenir une solution optimale.

IV.6.e. Comparaison avec l'algorithme de Fisher, Baruah et Baker

Tout d'abord, cet algorithme est comparé avec un autre algorithme, l'algorithme FBB-FFD, qui est un algorithme développé par Fisher, Baruah et Baker (cf. section II), qui permet d'obtenir un placement et un ordonnancement pour un système de tâches indépendantes, simultanées, à échéance inférieure ou égale à la période.
Une simulation a été réalisée, en générant aléatoirement 50 configurations différentes de tâches indépendantes, à échéance inférieure ou égale à la période. Le nombre de processeurs obtenu pour chaque cas est visible figure 3.12. Chaque configuration est composée de 100 tâches et le facteur d'utilisation total des tâches est de 10.
La recherche de solution est interrompue au bout de 5 minutes de calculs, et la solution utilisée est la dernière solution trouvée.
Comme nous pouvons le constater, l'AMNP donne un résultat utilisant 1 à 2 processeurs de moins que l'algorithme FBB-FFD. D'où un gain évident.
Par contre, il faut tout de même souligner que le temps de calcul pour l'algorithme de Fisher, Baruah et Baker est de l'ordre de la centième de seconde, sans optimisation, alors que le temps de calcul d'AMNP est de l'ordre de la minute.

IV.6.f. Conclusion

La comparaison de notre algorithme avec celui de Fisher, Baruah et Baker montre bien l'intérêt de la méthode développée. De plus, il ne faut pas oublier que cet algorithme peut s'employer dans un cadre relativement général de tâches avec contraintes de précédence, partage de ressources, aux échéances arbitraires.

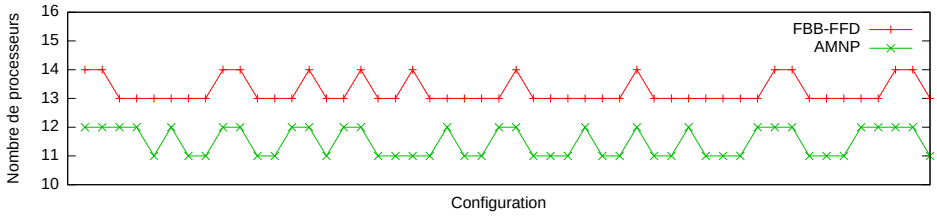


FIGURE 3.12 – Comparaison de l'algorithme développé avec l'algorithme FBB-FFD

L'inconvénient majeur de cette méthode est le temps de calcul. Toutefois, les calculs sont fortement parallélisables. Il est possible de paralléliser en découpant l'arbre de recherche en plusieurs sous-arbres, notamment au premier niveau de l'arbre, voire même au second niveau si nécessaire. L'utilisation d'un réseau d'ordinateurs ou de clusters de calcul pour la résolution d'un tel système accélérerait alors grandement les calculs.

IV.7. Optimisations envisagées

Une méthode envisageable, et qui peut paraître paradoxale, serait d'ajouter des contraintes. En effet, l'ajout de contraintes permettrait d'éliminer nombre de solutions de l'arbre de recherche et diminuer ainsi le temps nécessaire à son exploration.

IV.7.a. Prise en compte de la mémoire

Description du problème Souvent, les algorithmes ne prennent pas en compte les contraintes de mémoire. Dans notre cas, il serait assez simple d'en tenir compte. Les règles qu'il faudrait ajouter pour gérer la mémoire sont les mêmes que pour la gestion du facteur d'utilisation.
Le principe de gestion de la mémoire et du facteur d'utilisation est le même. La seule différence est que pour un processeur, le taux d'utilisation maximal autorisé est 1, alors que la mémoire totale disponible dépend uniquement du matériel.
Puisque dans notre modélisation nous nous intéressons à un pool de processeurs, tous les processeurs doivent être identiques, impliquant qu'ils disposent de la même quantité de mémoire.

En effet, le but de l'AMNP est de déterminer le nombre optimal de processeurs pour pouvoir ordonnancer un système (selon l'analyse holistique). S'il n'y a pas d'autres contraintes sur le système pour minimiser le nombre de processeurs et si plusieurs types de processeurs sont disponibles, chacun avec une quantité de mémoire différente alors le processeur muni de la plus grande mémoire sera choisi, car lui seul permettra d'atteindre le but recherché.
Pour pouvoir gérer plusieurs types de processeurs, il faudrait rajouter des contraintes, par exemple, des contraintes de coûts.

Règles à ajouter
- Le nombre minimum de processeurs doit être calculé en tenant compte des contraintes de mémoire. Il faut en effet que la mémoire totale disponible pour le pool soit supérieure ou égale à la somme de la mémoire requise pour toutes les tâches ;
- Une tâche ne peut être ajoutée à un processeur que si la quantité de mémoire encore libre est supérieure ou égale à la quantité de mémoire nécessaire à la tâche.

IV.7.b. Prise en compte de contraintes de placement des tâches

Description du problème Lors de la réalisation d'un système, il n'est pas rare d'avoir un capteur d'une part, et une tâche chargée de récupérer les informations issues de ce capteur d'autre part. Il est alors souhaitable d'avoir la tâche gérant le capteur s'exécutant sur un processeur lui-même proche du capteur, et non sur un processeur situé à l'opposé du capteur.
Il serait donc attrayant d'avoir la possibilité de spécifier un placement des tâches, par exemple, en précisant que deux tâches données doivent être placées sur le même processeur ou, a contrario, sur des processeurs différents.

Mise en place Avec quelques modifications l'AMNP est capable de tenir compte de cette contrainte supplémentaire. Le problème qui peut se poser est que, ne connaissant pas le nombre de processeurs nécessaires à l'ordonnancement du système de tâches, comment pré-affecter certaines tâches à un processeur ?
La réponse est relativement simple. A chaque tâche que l'on veut pré-affecter, on associe un numéro de processeur, notée N_{τ_i}. De même, à chaque processeur on associe un numéro, notée N_{P_i}, qui est initialisé à 0 au début de la recherche. Pendant la recherche, plusieurs cas peuvent alors se poser :
- La tâche τ_i à placer n'a pas de processeur prédéfini. Dans ce cas, la tâche est placée de manière classique.
- La tâche τ_i à placer a un processeur prédéfini. Dans ce cas, il faut encore distinguer deux cas de figure :
  - Le processeur sur lequel la tâche va être placée vérifie $N_{P_i} = 0$. Dans ce cas, le numéro du processeur devient le numéro du processeur associé à la tâche, et donc $N_{P_i} = N_{\tau_i}$.
  - Le processeur sur lequel la tâche va être placée vérifie $N_{P_i} \neq 0$. Dans ce cas, la tâche n'est placée que si la relation $N_{P_i} = N_{\tau_i}$ est vérifiée.

V. Conclusion

Dans ce chapitre, nous avons présenté un algorithme permettant de trouver une solution optimale vis-à-vis de l'analyse holistique. Comme nous avons pu le constater, la solution obtenue ne sera optimale qu'à la condition que notre algorithme s'exécute complètement, sans être interrompu. Introduire une limite de temps permet alors à l'utilisateur d'utiliser notre algorithme en tant qu'heuristique. De plus, nous avons constaté expérimentalement, que :
– le nombre de processeurs utilisé est alors proche du nombre optimal ;
– le temps nécessaire pour trouver la solution qui sera retournée est en général très court.
La solution fournie est alors valide, dans le sens où toutes les contraintes sont respectées : échéances, contraintes de précédence, communications, ressources partagées.
Nous espérons également que l'ajout de nouvelles contraintes permettra à notre algorithme de s'exécuter plus vite, notamment en permettant de réaliser plus de coupes dans l'arbre de recherche.

Publications Nos travaux ont fait l'objet de deux publications qui ont été acceptées à des conférences avec comité de sélection :
– l'une a été acceptée à la conférence internationale *RTNS'08*[1] [DRGR08] ;
– l'autre publication a été acceptée à la conférence francophone *ROADEF'09*[2] [DRGR09].

1. *Real-Time and Network Systems*
2. *Recherche Opérationnelle et d'Aide à la Décision*

Chapitre

4 Modèle à criticité multiple

Sommaire

Résumé

Ce chapitre présente nos travaux concernant un nouveau modèle de tâches récemment introduit par Vestal et qui prend en compte la notion de criticité.

I. Introduction

Une des briques fondamentales sur laquelle repose la validation des systèmes temps réel est la notion de temps d'exécution. En effet, les méthodes de validation que nous avons vues au chapitre 2 utilisent toutes cette notion. Par conséquent, une mauvaise évaluation du pire temps d'exécution d'une tâche peut conduire à valider un système qui est en fait non ordonnançable. La détermination du pire temps d'exécution est une opération difficile qui a reçu, et reçoit encore, beaucoup d'attention dans la littérature [WEE+08].
Schématiquement, il existe deux grandes méthodes :

- une méthode par *mesure de temps de réponse* : la tâche étudiée est exécutée un certain nombre de fois, et le temps d'exécution de chaque instance est mesuré. Le pire temps d'exécution sera alors le plus grand temps mesuré. Par le principe même de cette méthode, cette technique n'offre qu'une borne inférieure du pire temps d'exécution et ne peut être utilisée telle quelle en pratique (pour des systèmes temps réel dur tout du moins).
- une méthode *statique d'analyse de code source* : le code source de la tâche est analysé pour essayer d'en déduire son pire temps d'exécution. Mais cette méthode est difficile à mettre en œuvre, puisque il est nécessaire :
  - de connaître avec précision le processeur sur lequel la tâche sera exécutée (type, vitesse)
  - de déterminer, au travers des structures de contrôle du flux d'exécution (boucle if... then... else..., boucle for, etc...) quel sera le chemin menant au pire temps d'exécution.

  Cette méthode introduit généralement un pessimisme important [LBJ+95, TSH+03, SLPH+05].

Une détermination fine du temps d'exécution des tâches requiert donc du temps et de l'énergie, ce qui n'est pas sans coûts.
Nous avons donc ici deux objectifs qui s'opposent :

- réduire les coûts de développement des systèmes temps réels (et donc ici, les coûts de détermination de pire temps d'exécution des tâches) ;
- ne pas utiliser des pires temps d'exécution trop pessimistes pour analyser les tâches non critiques, puisque cela conduirait à un surdimensionnement du système (et donc à augmenter les coûts).

Il s'agit donc de trouver un juste milieu, et deux approches peuvent être utilisées :

- la première, est d'autoriser un certain nombre de dépassements du pire temps d'exécution (dûs à une approximation optimiste de ces derniers, par exemple). Certains modèles permettent la prise en compte de ce genre de problème. Nous pouvons citer les travaux de Bougueroua, qui a introduit la notion d'"allowance"[1] pour atteindre cet objectif ([Bou07]).
- la deuxième approche qui peut être utilisée est de considérer plusieurs niveaux de confiance pour le pire temps d'exécution. Une tâche nécessitant un haut degré de confiance ne devra jamais rater son échéance, tandis qu'une tâche nécessitant un degré de confiance moindre pourra manquer quelques échéances sans grandes conséquences sur le système. Ainsi, le pire temps d'exécution pour des tâches à haut degré de confiance devra être évalué avec la plus

1. Bougueroua utilise le terme allowance aussi bien en français qu'en anglais.

Niveau	Condition	Description
A	Catastrophique	Une défaillance peut provoquer un crash.
B	Dangereuse	Une défaillance a un large impact négatif sur la sécurité ou les performances, ou réduit les possibilités de l'équipage à gérer l'avion à cause de souffrances physiques ou une charge de travail trop importante, ou provoque des blessures sérieuses voir fatales blessures parmi les passagers.
C	Majeur	Une défaillance est significative, mais a moins d'impact qu'une défaillance dangereuse (par exemple, provoque un inconfort des passagers plutôt que des blessures).
D	Mineur	Une défaillance est notable, mais a moins d'impact qu'une défaillance majeure (par exemple, provoquant des désagréments auprès des passagers ou un changement de plan de vol).
E	Sans effet	Une défaillance n'a pas d'impact sur la sécurité, les opérations ou la charge de travail de l'équipage.

TABLE 4.1 – Niveaux de criticité dans la norme DO-178B

grande précision possible parce qu'une sous-estimation peut provoquer un non respect d'une échéance, ce qui peut avoir des conséquences catastrophiques pour le système, et une sur-estimation peut conduire à la non validation d'un système alors que toutes les échéances sont respectées. L'idée est donc de procéder à une estimation fine du pire temps de réponse pour les tâches nécessitant un haut degré de confiance, et de permettre une évaluation plus "approximative" pour les tâches ne nécessitant qu'un faible degré de confiance.

Certains standards aéronautiques définissent plusieurs niveaux de criticité précisant le niveau de confiance requis. Ainsi, le standard DO-178B [ARI97] définit 5 niveaux de criticité, notés de A à F. Une défaillance (ie. le non respect d'une échéance) d'une tâche de criticité A peut avoir des conséquences catastrophiques sur le système tout entier (par exemple, crash d'un avion), tandis qu'une défaillance d'une tâche de criticité F n'a pas de conséquences notables sur le système. Le tableau 4.1 résume les conséquences d'une défaillance d'une tâche en fonction de son niveau de criticité.

Un moyen pour tenir compte de ces différents niveaux de criticité est de réaliser un partitionnement temporel entre les différentes applications logicielles, ce qui permet de réaliser une isolation temporelle comme décrit dans le standard ARINC 653 ([ARI97]). Le standard ARINC 653 définit une API [2] qui fournit un partitionnement temporel pour les applications nécessitant des niveaux de confiance différents. La ligne du temps est définie comme un ensemble de partitions, chaque partition ayant un certain quota de temps déterminé. Chaque tâche (ou ensemble de tâches dépendantes) est attachée à une partition et un algorithme d'ordonnancement classique est exécuté dans chaque partition. Puisque chaque partition a un certain quota

2. Application Programming Interface

de temps prédéterminé, une partition ne peut interférer avec une autre. En d'autres termes, une tâche appartenant à une partition A ne peut interférer avec une tâche appartenant à une partition B. Ainsi, en affectant toutes les tâches nécessitant le même niveau de confiance à une même partition, il est possible d'assurer une isolation temporelle entre les tâches nécessitant des niveaux de confiance différents.
Un autre moyen de tenir compte des niveaux de criticité a été introduit par Vestal dans un article récent [Ves07]. Vestal a introduit un nouveau modèle pour représenter les systèmes de tâches. Ce modèle se base sur la considération de plusieurs pire temps d'exécution au lieu d'un seul, ce qui permet de moduler la confiance en fonction de la criticité de la tâche. Baruah et Vestal donne cette définition dans [BV08] : "the more confidence one needs in a task execution time bound, the larger and more conservative that bound tends to be in pratice"[3].
Le modèle de tâches à criticité multiple permet de tenir compte du niveau de criticité d'une tâche (est-elle critique ou non ?) lors de la détermination de son temps d'exécution. Ainsi, la détermination du pire temps d'exécution d'une tâche non critique se fera par une analyse "rapide", tandis que la détermination du pire temps d'exécution d'une tâche critique se fera par une analyse "fine" (et donc coûteuse).
Mais se contenter de cette détermination du pire temps de réponse en fonction de la criticité de la tâche n'est pas suffisante, puisqu'un risque existant est qu'une tâche non critique, ratant son échéance du fait d'un temps d'exécution erroné, fasse rater son échéance à une tâche critique. Vestal répond à ce problème à travers le modèle qu'il a introduit et que nous allons présenter dans le paragraphe suivant.

I.1. Modèle à criticité multiple

Le modèle développé par Vestal dans [Ves07] est basé sur le modèle classique de Liu et Layland [LL73]. Le modèle de Liu et Layland ayant déjà été présenté dans le cadre de cette thèse (cf. chapitre 1 section II.2.c), nous ne définirons que les différences entre le modèle de Liu et Layland et le modèle de Vestal :

- introduction d'un nouveau paramètre correspondant au niveau de criticité L_i de la tâche τ_i, qui spécifie le niveau de sûreté requis pour la tâche τ_i. Par convention, nous supposons que le niveau 1 représente le plus petit niveau de criticité.
- le pire temps d'exécution C_i est remplacé par une fonction $C_i : N^+ \rightarrow R+$, qui spécifie le pire temps d'exécution pour les différents niveaux de criticité. Nous pouvons donc noter que C_i n'est plus une constante mais est une fonction dépendant du niveau de criticité. Aussi, dans le cadre du modèle à criticité multiple, le pire temps d'exécution d'une tâche τ_i pour le niveau de criticité ℓ sera noté par $C_i(\ell)$.

La notion de facteur d'utilisation d'une tâche est également modifiée, puisque chaque tâche dispose maintenant de plusieurs temps d'exécution. Elle peut toutefois être définie en utilisant

3. Plus on a besoin de confiance pour le temps d'exécution d'une tâche, plus grande et plus conservative elle tend à être en pratique

	$D_i = T_i$	π_i	L_i	$C_i(1)$	$C_i(2)$
τ_1	4	0	2	2	2
τ_2	7	1	1	2	5

TABLE 4.2 – Exemple de système à criticité multiple

	$D_i = T_i$	π_i	C_i
τ_1	4	0	2
τ_2	7	1	2

TABLE 4.3 – Système de tâches à considérer lors de l'étude de la tâche τ_2

le temps d'exécution correspondant au niveau de criticité L_i de la tâche τ_i :

$$u_i \stackrel{\text{def}}{=} \frac{C_i(L_i)}{T_i} \tag{4.1}$$

et le facteur d'utilisation du système reste défini comme étant la somme des facteurs d'utilisation des tâches le composant.
Nous avons dit qu'une tâche avait un temps d'exécution par niveau de criticité. Ces temps d'exécutions sont reliés par la relation suivante :

$$C_i(\ell) \leq C_i(\ell + 1) \tag{4.2}$$

L'utilisation de ce modèle est relativement simple : lors de l'étude d'une tâche τ_i, ne sont considérés que les temps d'exécution correspondant au niveau de criticité L_i de la tâche τ_i. Par exemple, pour le système de tâches à criticité multiple dont les caractéristiques sont résumées table 4.2, lors de l'étude la tâche τ_2 (par exemple, pour déterminer son temps de réponse), nous ne considérerons que les temps d'exécution correspondant à son niveau de criticité, c'est-à-dire 1. Ainsi, l'étude de la tâche τ_2 revient à considérer le système de tâches classiques dont les caractéristiques sont données table 4.3.
Nous pouvons donc constater que l'utilisation de ce modèle reste relativement simple.

I.2. Système sporadique équivalent

Dans [BV08], Baruah et Vestal ont introduit la notion de système de tâches sporadiques équivalent :

Définition 15. *A chaque tâche à criticité multiple τ_i est définie une tâche sporadique correspondante $\tau_i'(C_i(L_i), D_i, T_i)$.*

Cette notion leur permet d'introduire le théorème suivant :

Théorème 26. *Un système de tâches à criticité multiple est ordonnançable si, et seulement si, le système de tâches sporadiques équivalent est ordonnançable.*

	$D_i = T_i$	L_i	$C_i(1)$	$C_i(2)$
τ_1	5	1	1	2
τ_2	5	2	2	5

TABLE 4.4 – Exemple de système à criticité multiple

Toutefois, pour que ce théorème soit juste, il manque une hypothèse dans [BV08]. En effet, il est nécessaire de tenir compte de la contrainte suivante :

$$\forall i \in [1, n], \forall j \in [1, L_i], C_i(j) = C_i(L_i) \tag{4.3}$$

Sans cette hypothèse, le théorème 26 est faux. Il suffit pour le démontrer de considérer le système de tâches dont les caractéristiques sont résumées table 4.4. Une analyse du système de tâches sporadiques classique montre que ce système n'est pas ordonnançable, puisque le facteur d'utilisation de ce système est supérieur à 1 :

$$\frac{C_1(L_1)}{T_1} + \frac{C_2(L_2)}{T_2} = 1.2 > 1 \tag{4.4}$$

Par contre, d'un point de vue criticité multiple, ce système de tâches est ordonnançable en utilisant des priorités fixes lorsque la priorité la plus forte est donnée à la tâche τ_2. Pour s'en convaincre, il suffit de calculer les temps de réponses des tâches τ_1 et τ_2 :

$$Tr_1 = C_1(L_1) = 5 \leq D_1 \tag{4.5}$$
$$Tr_2 = C_1(L_2) + C_2(L_2) = 3 \leq D_2 \tag{4.6}$$

Ainsi, toutes les échéances semblent respectées, ce qui est impossible si le système est surchargé.

II. Algorithme de Vestal

II.1. Présentation

Dans [Ves07], Vestal a introduit une version modifiée de l'algorithme d'Audsley [Aud91]. L'algorithme de Vestal est optimal pour l'ordonnancement de systèmes de tâches indépendantes à échéance contrainte [BV08].
Comme nous l'avons vu dans le premier chapitre de cette thèse, l'algorithme d'Audsley se base sur l'observation suivante : le temps de réponse d'une tâche dépend uniquement de l'ensemble des tâches plus prioritaires. Il n'est pas nécessaire de connaître une affectation des priorités précise. Ainsi, le principe de l'algorithme d'Audsley est d'affecter une tâche à chaque niveau de priorité, en commençant par le niveau de priorité le plus faible, et la tâche qui est affectée à un niveau de priorité donné est la première tâche trouvée qui respecte son échéance pour ce niveau de priorité.

Vestal a modifié la façon dont la tâche à affecter à un niveau de priorité est choisie : au lieu d'affecter la première tâche trouvée qui peut être ordonnancée à ce niveau de priorité, Vestal a introduit une méthode de sélection basée sur la notion de *facteur d'échelle critique*[4], c'est-à-dire le facteur maximum par lequel la vitesse du processeur peut être réduit sans nuire à l'ordonnançabilité de la tâche τ_i. A chaque niveau de priorité, la tâche qui sera assignée sera la tâche ayant le plus grand facteur d'échelle critique pour ce niveau de priorité.
Nous rappelons la définition du facteur d'échelle critique d'un système donné par Lehoczky dans [LSD89] :

$$\Delta^* \stackrel{\text{def}}{=} \left[\max_{1 \leq i \leq n} \min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \right]^{-1} \tag{4.7}$$

où S_i est l'ensemble des points d'ordonnancement définit dans [LSD89] par :

$$S_i \stackrel{\text{def}}{=} \left\{ kT_j | j = 1, \ldots, i; k = 1, \ldots, \left\lfloor \frac{D_i}{T_j} \right\rfloor \right\} \tag{4.8}$$

Le facteur d'échelle critique d'une tâche se définit simplement par :

$$\Delta_i \stackrel{\text{def}}{=} \left[\min_{t \in S_i} \sum_{j=1}^{i} \frac{C_j}{t} \left\lceil \frac{t}{T_j} \right\rceil \right]^{-1} \tag{4.9}$$

Et donc, nous avons la relation suivante entre le facteur d'échelle critique Δ^* d'un système de tâches et celui des tâches Δ_i :

$$\Delta^* = \min_{1 \leq i \leq n} (\Delta_i) \tag{4.10}$$

Nous donnons figure 4.1 un exemple d'utilisation de l'algorithme d'assignation des priorités de Vestal. Le tableau supérieur résume les caractéristiques des tâches, tandis que le tableau inférieur est la trace de l'algorithme lui-même. Par exemple, lorsque nous essayons d'assigner une tâche au plus faible niveau de priorité (c'est-à-dire le niveau 3), nous calculons le facteur d'échelle critique de chaque tâche, et nous choisissons celle ayant le plus grand, c'est-à-dire ici la tâche τ_3. Nous continuons le processus en étudiant le niveau de priorité 2, sans oublier de retirer la tâche 3, puisqu'elle a été affectée à la priorité 3. La tâche ayant le plus grand facteur d'échelle critique est ici la tâche τ_0, aussi, nous assignons τ_0 à ce niveau de priorité. Le processus continue jusqu'à ce que toutes les tâches soient affectées d'une priorité.
Le facteur d'échelle critique du système est alors le minimum des facteurs d'échelle critiques de chaque tâche à son niveau de priorité, c'est-à-dire, ici $\Delta = \min(1.69461, 3.86957, 2.2, 11) = 1.69461$, le facteur d'échelle critique de la tâche τ_3.

4. *critical scaling factor* en anglais

τ_i	T_i	D_i	L_i	$C_i(1)$	$C_i(2)$
0	164	104	1	7	17
1	89	44	2	4	4
2	191	80	1	12	16
3	283	283	2	85	85

Priorité	Trace
3	$\left.\begin{array}{l}\Delta_{\tau_0} = 0.928571\\ \Delta_{\tau_1} = 0.360656\\ \Delta_{\tau_2} = 0.740741\\ \Delta_{\tau_3} = 1.69461\end{array}\right\} \Rightarrow \pi_3 = 3$
2	$\left.\begin{array}{l}\Delta_{\tau_0} = 3.86957\\ \Delta_{\tau_1} = 1.18919\\ \Delta_{\tau_2} = 3.47826\end{array}\right\} \Rightarrow \pi_0 = 2$
1	$\left.\begin{array}{l}\Delta_{\tau_1} = 2.2\\ \Delta_{\tau_2} = 5\end{array}\right\} \Rightarrow \pi_2 = 1$
0	$\left.\Delta_{\tau_1} = 11\right\} \Rightarrow \pi_1 = 0$

$\Delta = min_{\Delta_i} = 1.69461$

FIGURE 4.1 – Trace de l'assignation des priorités selon l'algorithme de Vestal

II.2. Optimalité de l'algorithme d'Audsley

Dans [BV08], les auteurs précisent que l'algorithme de Vestal est optimal, sans pour autant le démontrer. L'algorithme de Vestal étant un cas particulier de l'algorithme d'Audsley, si nous pouvons montrer l'optimalité de l'algorithme d'Audsley, alors nous montrerons, par la même occasion, l'optimalité de l'algorithme de Vestal.

Théorème 27. *L'algorithme d'assignation des priorités d'Audsley est optimal pour l'ordonnancement de systèmes de tâches indépendantes à échéance contrainte et à criticité multiple en priorité fixe.*

Pour démontrer ce résultat, nous allons avoir besoin de 2 résultats intermédiaires, synthétisés par les lemmes suivants.

Lemme 1. *Lorsque nous étudions une tâche τ_i, nous pouvons considérer le système de tâches sporadiques équivalent au lieu du système à criticité multiple, en prenant en compte les temps d'exécutions correspondant au niveau de criticité L_i de la tâche étudiée τ_i.*

Démonstration. Ce lemme peut être déduit de la définition d'un système de tâches à criticité multiple. Lorsque nous calculons le pire temps de réponse de la tâche τ_i, nous ne considérons que les pire temps d'exécution correspondant au niveau de criticité de la tâche τ_i, comme nous pouvons le voir dans l'équation 4.11, qui est une version modifiée de l'équation établie par

Joseph et Pandia dans [JP86] et remaniée par Vestal dans [Ves07] pour calculer le pire temps de réponse des tâches à criticité multiple :

$$Tr_i = \sum_{j=1}^{i} \left\lceil \frac{Tr_i}{T_j} \right\rceil C_j(L_i) \tag{4.11}$$

Ainsi, lorsque nous étudions la tâche τ_i, nous pouvons seulement considérer le système de tâches classiques équivalent correspondant au système de tâches en ne prenant en compte que les temps d'exécution correspondant à la criticité L_i de la tâche étudiée τ_i. □

Si nous regardons avec attention le système de tâches illustré en figure 4.1, nous pouvons constater que le facteur d'échelle critique de la tâche τ_1, lorsqu'elle est affectée au niveau de priorité 2, est plus grand grand que lorsqu'elle est assignée au niveau de priorité 3 (c'est-à-dire à un niveau de plus faible priorité). Ce résultat intuitif est l'objet du second lemme :

Lemme 2. *Soit τ_i une tâche avec un facteur d'échelle critique $\Delta_{i,j}$ lorsqu'elle est assignée à la priorité j ; si τ_i est assignée à la priorité $j-1$, alors le facteur d'échelle critique de τ_i vérifiée $\Delta_{i,j} < \Delta_{i,j-1}$.*

Démonstration. Dans le cadre de cette preuve, nous ne considérerons que la tâche τ_i que nous assignerons au niveau de priorité j ou $j-1$. Il est important de noter que la seule différence entre ces deux affectations est l'ensemble des tâches plus prioritaires. Lorsque la tâche τ_i est affectée à la priorité j, l'ensemble des tâches plus prioritaires contient une tâche supplémentaire par rapport à l'ensemble des tâches plus prioritaires lorsque la tâche est affectée à la priorité $j-1$. Par convention (et sans perte de généralité, puisque nous ne faisons aucune hypothèse quant à l'ordre des tâches plus prioritaires), nous supposerons que cette tâche est la tâche τ_j. Lehoczky définit le facteur d'échelle critique dans [Leh90] :

$$\Delta_{i,j} \stackrel{\text{def}}{=} \left[\min_{t \in S_{i,j}} \frac{1}{t} \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t}{T_k} \right\rceil \right]^{-1} \tag{4.12}$$

$$\Delta_{i,j-1} \stackrel{\text{def}}{=} \left[\min_{t \in S_{i,j-1}} \frac{1}{t} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t}{T_k} \right\rceil \right]^{-1} \tag{4.13}$$

Ces définitions ont juste été adaptées aux systèmes de tâches à criticité multiple, en remplaçant le temps d'exécution classique C_k par le temps d'exécution $C_k(L_i)$, qui est le temps d'exécution de la tâche τ_k au niveau de criticité L_i de la tâche étudiée τ_i.
$S_{i,j}$ représente l'ensemble des points d'ordonnancement pour la tâche τ_i lorsque τ_i est assignée à la priorité j. Cet ensemble est défini par l'équation suivante :

$$S_{i,j} \stackrel{\text{def}}{=} \left\{ kT_m | m = 1, \ldots, j; k = 1, \ldots, \left\lfloor \frac{D_i}{T_m} \right\rfloor \right\} \cup \{D_i\} \tag{4.14}$$

Bien que Bini et al. aient introduit dans [BB04b] un sous-ensemble suffisant de point d'ordonnancement, nous avons besoins de considérer, dans le cadre de notre preuve, tous les points d'ordonnancement.
Aussi, d'après les équations 4.12 et 4.13, il existe t_j et t_{j-1} tels que :

$$\Delta_{i,j} = \left[\frac{1}{t_j} \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil \right]^{-1} \tag{4.15}$$

$$\Delta_{i,j-1} = \left[\frac{1}{t_{j-1}} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_{j-1}}{T_k} \right\rceil \right]^{-1} \tag{4.16}$$

Nous pouvons remarquer que $S_{i,j-1} \subset S_{i,j}$. Nous avons donc 2 cas à prendre en compte : $t_j \in S_{i,j-1}$ et $t_j \notin S_{i,j-1}$:
– Si $t_j \in S_{i,j-1}$. Il est alors évident que :

$$\forall t, \frac{1}{t} \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t}{T_k} \right\rceil > \frac{1}{t} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t}{T_k} \right\rceil \tag{4.17}$$

Si $t = t_j$ alors :

$$\frac{1}{t_j} \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil > \frac{1}{t_j} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil \tag{4.18}$$

Puisque $t_j \in S_{i,j-1}$ et t_{j-1} minimise $\frac{1}{t} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t}{T_k} \right\rceil$ (voir la définition de t_{j-1}, équation 4.16), nous avons :

$$\frac{1}{t_{j-1}} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_{j-1}}{T_k} \right\rceil \leq \frac{1}{t_j} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil \tag{4.19}$$

Les équations 4.18 et 4.19 nous donne :

$$\frac{1}{t_{j-1}} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_{j-1}}{T_k} \right\rceil < \frac{1}{t_j} \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil \tag{4.20}$$

C'est-à-dire :

$$\Delta_{i,j-1} > \Delta_{i,j} \tag{4.21}$$

– Maintenant, si nous considérons le cas $t_j \notin S_{i,j-1}$.
Par définition, nous pouvons noter que $D_i = \max(S_{i,j})$ et $D_i = max(S_{i,j-1})$. Puisque $t_j \notin S_{i,j-1}$, nous avons $t_j \neq D_i$. Ainsi :

$$\exists t_k \in S_{i,j-1}, t_j < t_k \tag{4.22}$$

Nous pouvons également noter que $\sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t}{T_k} \right\rceil$ est une fonction constante par morceaux et que t_j n'est pas un point de discontinuité puisque $t_j \notin S_{i,j-1}$, ainsi :

$$\begin{array}{l} \exists t_k \in S_{i,j-1}, \\ \begin{cases} t_k > t_j \\ \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil = \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_k}{T_k} \right\rceil \end{cases} \end{array} \tag{4.23}$$

De plus :

$$\sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil < \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil \tag{4.24}$$

Et donc, les équations 4.23 et 4.24 nous conduisent à :

$$\sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_k}{T_k} \right\rceil < \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil \tag{4.25}$$

Puisque $t_k > t_j$, nous avons $\frac{1}{t_k} < \frac{1}{t_j}$. Et si nous utilisons l'équation 4.25, nous avons :

$$\frac{1}{t_k} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_k}{T_k} \right\rceil < \frac{1}{t_j} \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil \tag{4.26}$$

Par définition de t_{j-1} (i.e., équation 4.16), nous avons :

$$\frac{1}{t_{j-1}} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_{j-1}}{T_k} \right\rceil = \min_{t \in S_{i,j-1}} \frac{1}{t} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t}{T_k} \right\rceil \tag{4.27}$$

Et donc, puisque $t_k \in S_{i,j-1}$:

$$\frac{1}{t_{j-1}} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_{j-1}}{T_k} \right\rceil \leq \frac{1}{t_k} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_k}{T_k} \right\rceil \tag{4.28}$$

Si nous combinons les équations 4.26 et 4.28, nous obtenons :

$$\frac{1}{t_{j-1}} \sum_{k=1}^{j-1} C_k(L_i) \left\lceil \frac{t_{j-1}}{T_k} \right\rceil < \frac{1}{t_j} \sum_{k=1}^{j} C_k(L_i) \left\lceil \frac{t_j}{T_k} \right\rceil \tag{4.29}$$

C'est-à-dire :

$$\Delta_{i,j-1} > \Delta_{i,j} \tag{4.30}$$

Nous avons donc prouvé dans les deux cas ($t_j \in S_{i,j-1}$ et $t_j \notin S_{i,j-1}$) que $\Delta_{i,j-1} > \Delta_{i,j}$. Ce qui prouve le lemme. □

Nous pouvons donc maintenant prouver le théorème 27

Démonstration du théorème 27. En utilisant le lemme 1, étudier l'ordonnançabilité d'une tâche à criticité multiple peut être fait en étudiant l'ordonnançabilité d'un système de tâches équivalent correspondant au niveau de criticité de la tâche étudiée. Et en prenant en compte le lemme 2, le facteur d'échelle critique d'une tâche ne peut qu'augmenter lorsqu'une tâche est assignée à un niveau de priorité plus élevé. Puisque les hypothèses du modèle de tâches classiques sont respectées dans le cadre du modèle à criticité multiple, nous pouvons en déduire que l'algorithme d'Audsley est également optimal pour des systèmes à criticité multiple. □

Ayant ce théorème, nous pouvons en déduire le théorème suivant :

Théorème 28. *L'algorithme de Vestal est optimal pour ordonnancer un ensemble de tâches à échéance contrainte et à criticité multiple en priorité fixe.*

Démonstration. Puisque l'algorithme de Vestal est un cas particulier de l'algorithme d'Audsley, l'optimalité de l'algorithme de Vestal découle de l'optimalité de l'algorithme d'Audsley. □

II.3. Vitesse du processeur

II.3.a. Maximisation du facteur d'échelle critique

Pour les systèmes de tâches à criticité multiple, l'algorithme d'Audsley est optimal. Mais si le système n'est pas ordonnançable, alors déterminer le facteur d'accélération minimum pour la vitesse du processeur afin de rendre le système ordonnançable en utilisant un algorithme à priorité fixe est une question importante pour les concepteurs de systèmes temps réel.
Pour les tâches sporadiques à échéance contrainte, l'assignation des priorités (i.e., Deadline Monotonic) et la détermination du facteur d'accélération sont des problèmes indépendants. Nous avons prouvé que ce résultat était toujours vrai pour des systèmes de tâches à criticité multiple et que, de plus, les deux problèmes pouvaient être résolus simultanément (c'est-à-dire que le facteur d'accélération peut être déterminé en utilisant un algorithme glouton [5] pendant l'assignation des priorités).
L'algorithme 1 représente une implémentation de notre algorithme en pseudo-code. Il détermine une assignation des priorités et un facteur d'échelle critique Δ^*. La fonction $\Delta(\tau_i, hep(\tau_i))$ calcule le facteur d'échelle critique de la tâche τ_i, où $hep(\tau_i)$ est l'ensemble des tâches de priorité supérieure ou égale à τ_i.
Si le facteur d'échelle critique Δ^* est plus grand que 1, alors cela correspond au facteur de réduction maximum par lequel nous pouvons diviser la vitesse du processeur sans nuire à l'ordonnançabilité du système. Si Δ^* est plus petit que 1, alors le système de tâches n'est pas ordonnançable et Δ^* correspond au facteur d'accélération à appliquer à la vitesse du processeur afin de rendre le système ordonnançable.

5. Un algorithme glouton est un algorithme qui suit le principe de faire, étape par étape, un choix optimum local dans l'espoir d'obtenir un résultat optimum global

Algorithme 1 : Modulation de la vitesse du processeur et assignation des priorités

Input : τ^* = ensemble de tâche à ordonnancer
Output : Δ^* = facteur d'échelle maximum
Output : $\tilde{\tau}$ = système de tâches ordonnancées
$\tau \Leftarrow \tau^*$ $\tilde{\tau} \Leftarrow \emptyset$ **for** *j from n to* 1 **do**
  $\tau_{\text{Vestal}} = \emptyset$ **for** $\tau_A \in \tau$ **do**
    if $\tau_{\text{Vestal}} = \emptyset$ **then**
      $\tau_{\text{Vestal}} \Leftarrow \tau_A$ $\Delta^* = \Delta(\tau_A, \tau)$
    else
      if $\Delta(\tau_{\text{Vestal}}, \tau) < \Delta(\tau_A, \tau)$ **then**
        $\tau_{\text{Vestal}} \Leftarrow \tau_A$
  $\pi(\tau_{\text{Vestal}}) \Leftarrow j$ $\tau \Leftarrow \tau - \{\tau_{\text{Vestal}}\}$ $\tilde{\tau} \Leftarrow \tilde{\tau} \cup \{\tau_{\text{Vestal}}\}$ **if** $\Delta(\tau_{\text{Vestal}}, \tau) < \Delta^*$ **then**
    $\Delta^* = \Delta(\tau_{\text{Vestal}}, \tau)$

Notre résultat principal, le théorème 29, se base sur le lemme suivant :

Lemme 3. *Soit τ un système de tâches et τ_i et τ_j deux tâches avec τ_i plus prioritaire que τ_j. Si le facteur d'échelle critique de la tâche τ_i au niveau de priorité de τ_j est plus grand que le facteur d'échelle critique de τ_j à son niveau de priorité, alors insérer la tâche τ_i au niveau de priorité de τ_j ne peut qu'augmenter le facteur d'échelle critique du système de tâches.*

Démonstration. Nous allons utiliser un argument basé sur une permutation pour prouver le résultat. La figure 4.2 représente la transformation effectuée. Chaque zone a une signification :
- La zone 1 est composée de tâches ayant une priorité plus grande que la tâche τ_i ;
- La zone 2 est composée de tâches ayant une priorité intermédiaire, c'est-à-dire plus prioritaire que la tâche τ_j mais moins prioritaire que la tâche τ_i ;
- La zone 3 est composée de tâches ayant une priorité plus faible que la tâche τ_j.

Si nous étudions l'évolution du facteur d'échelle critique de chaque tâche lorsque la transformation est effectuée, nous observons que :
- Le facteur d'échelle critique des tâches de la zone 1 est inchangé, puisque non influencé par la modification des priorités de tâches de priorités inférieures
- Le facteur d'échelle critique des tâches de la zone 3 est également inchangé, puisque non influencé par la modification des priorités de tâches de priorités supérieures. En effet, l'ensemble des tâches de priorités supérieures reste le même.
- Le facteur d'échelle critique des tâches de la zone 2 ne peut qu'augmenter d'après le lemme 2.

Enfin, si la transformation est effectuée, c'est que par hypothèse, la tâche τ_i a un facteur d'échelle critique au niveau de priorité de τ_j qui est plus grand que le facteur d'échelle critique de τ_j.
En d'autres termes, dans tous les cas, le facteur d'échelle critique de chaque tâche ne peut qu'augmenter ou être inchangé, hormis pour la tâche τ_i. Mais par hypothèse, le nouveau facteur

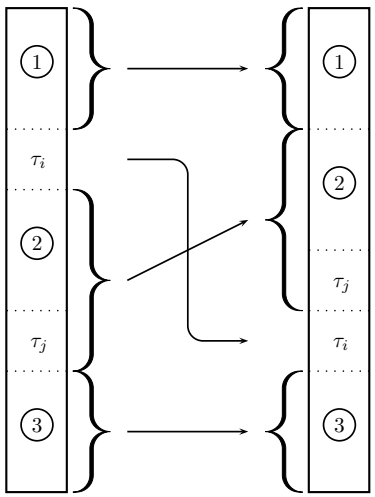


FIGURE 4.2 – Schéma de la transformation

d'échelle critique de la tâche τ_i est plus grand que le précédent facteur d'échelle critique de la tâche τ_j. Ainsi donc, le facteur d'échelle critique du système ne peut qu'augmenter. Au pire, il reste inchangé. □

Disposant maintenant de ce lemme, nous pouvons prouver le théorème suivant :

Théorème 29. *L'algorithme de Vestal retourne une assignation des priorités maximisant le facteur d'échelle critique, c'est-à-dire le facteur d'accélération minimum de la vitesse du processeur si le système n'est pas ordonnançable).*

Démonstration. Soit τ le système de tâches, composé de n tâches $\tau_1, \ldots, \tau_n$ et chaque tâche est affectée à un niveau de priorité. Pour prouver le résultat, nous allons reconstruire l'assignation des priorités de Vestal à partir de τ en utilisant le lemme 3. La méthode est simple : nous recherchons la tâche ayant le plus grand facteur d'échelle critique au niveau de priorité n parmi toutes les tâches ayant une priorité supérieure ou égale à n. Nous insérons alors cette tâche à ce niveau de priorité. Par le lemme 3, le facteur d'échelle critique de ce nouveau système de tâches τ' ne peut être que plus grand ou égal au facteur d'échelle critique de τ. Nous répétons alors l'opération, en remplaçant τ par τ' et en recherchant la tâche à insérer au niveau de priorité $n-1$, et ainsi de suite jusqu'à ce que le niveau de priorité soit égal à 1.
De cette manière, nous construisons un nouvel ordonnancement à partir de celui de départ. Le nouvel ordonnancement est le même que celui retourné par l'algorithme de Vestal, parce que, dans les deux cas, à chaque étape, la même tâche est sélectionnée. Puisque la transformation

charge	10 tâches		20 tâches		30 tâches	
	gain	ordonnancement identique	gain	ordonnancement identique	gain	ordonnancement identique
0.05	807.06%	0.11%	751.18%	0.00%	726.76%	0.00%
0.10	351.53%	0.44%	305.64%	0.04%	275.53%	0.00%
0.15	209.14%	1.32%	173.22%	0.23%	153.29%	0.14%
0.20	142.76%	2.46%	115.85%	0.85%	97.63%	0.68%
0.25	102.97%	3.99%	83.05%	1.82%	67.56%	1.19%
0.30	77.70%	5.56%	62.30%	2.68%	50.60%	2.04%
0.35	59.39%	7.41%	47.78%	3.92%	38.54%	2.85%
0.40	45.71%	8.86%	37.81%	4.72%	30.46%	3.03%
0.45	35.71%	11.26%	29.75%	5.76%	24.80%	4.25%
0.50	27.65%	12.96%	23.72%	7.45%	20.80%	5.01%
0.55	21.16%	16.16%	18.90%	8.52%	16.31%	6.65%
0.60	15.81%	19.66%	14.96%	9.31%	12.76%	6.44%
0.65	11.45%	23.94%	10.82%	13.21%	9.72%	8.66%
0.70	7.75%	30.16%	7.65%	14.27%	7.22%	10.06%
0.75	4.41%	41.63%	4.75%	19.40%	4.58%	12.13%
0.80	2.03%	57.83%	2.16%	34.56%	1.89%	27.24%
0.85	0.88%	70.58%	0.82%	55.98%	0.46%	58.33%
0.90	0.29%	82.03%	0.12%	71.42%	0.00%	100.00%
0.95	0%	100.00%	0.00%	100.00%	0.00%	100.00%

FIGURE 4.3 – Résultats des comparaisons

utilisée ne peut qu'augmenter le facteur d'échelle critique du système de tâches initial τ et que ce système est quelconque, nous pouvons conclure que l'affectation de priorité résultant de l'algorithme de Vestal maximise le facteur d'échelle critique pour les algorithmes à priorité fixe. Ceci prouve donc le théorème 29. □

Ainsi, l'algorithme de Vestal, en maximisant le facteur d'échelle critique, a un grand intérêt puisqu'il offre une manière simple de définir avec précision la vitesse minimum du processeur de sorte que le système soit ordonnançable.

II.3.b. Résultats expérimentaux

Dans le but de comparer les algorithmes d'Audsley et de Vestal, nous avons comparé les facteurs d'échelle critiques obtenus pour différentes configurations de tâches, et déterminé ainsi des statistiques. Voici un résumé de nos conditions expérimentales :
– le nombre de tâches est choisi dans l'ensemble $\{10, 20\}$;
– la charge est choisie dans l'ensemble $\{0.2, 0.4, 0.6, 0.8\}$;
– nous avons effectué 100 000 essais par configuration.
La colonne gain correspond au gain relatif, en pourcentage, du facteur d'échelle critique de l'ordonnancement retourné par l'algorithme de Vestal par rapport à celui retourné par l'algorithme

d'Audsley. Il correspond à la formule suivante :

$$pourcentage = \frac{\Delta_v - \Delta_a}{\Delta_a} * 100 \tag{4.31}$$

où Δ_a correspond au facteur d'échelle critique de l'ordonnancement retourné par l'algorithme d'Audsley et Δ_v au facteur d'échelle critique de celui retourné par l'algorithme de Vestal. De plus, comme nous avons montré que l'algorithme de Vestal maximisait le facteur d'échelle critique, nous sommes assurés que le résultat est positif.
Nous pouvons voir figure 4.4 que le gain est plus important lorsque le facteur d'utilisation des systèmes de tâches est faible. Ce résultat est logique et facile à comprendre : pour des systèmes de tâches à faible facteur d'utilisation, il existe un grand nombre de configurations ordonnançables, et l'algorithme d'Audsley en prend une arbitrairement. Aussi, il n'y a que peu de chance que la configuration retenue soit optimale ou proche de l'optimale d'un point de vue du facteur d'échelle critique.
Au contraire, pour des systèmes à haut facteur d'utilisation, il n'existe que peu de configurations valides, et les chances de choisir une configuration optimale ou proche de l'optimal sont donc plus grandes.
Ces résultats sont confirmés par le ratio des ordonnancements identiques retournés par les algorithmes de Vestal et d'Audsley :
- pour les systèmes faiblement chargés, ce ratio est très faible (moins de 1% pour 10 tâches et une charge de 0.2)
- pour les systèmes fortement chargés, ce ratio augmente fortement (plus de 50% pour 10 tâches et une charge de 0.8)

Nous avons également étudié l'influence que pouvait avoir le nombre de niveaux de criticité sur le gain (figure 4.5). Nous pouvons noter que ce nombre n'a qu'une faible influence, et n'est vraiment sensible que pour les systèmes de tâches faiblement chargés (charge inférieure à 20%), où, dans ce cas, augmenter le nombre de niveaux de criticité diminue le gain que peut avoir l'algorithme de Vestal sur celui d'Audsley.

III. Analyse de sensibilité

III.1. Introduction

La plupart des méthodes d'analyse d'ordonnançabilité souffre d'un inconvénient majeur : à la question "le système est-il ordonnançable ?", elles ne répondent que par oui ou par non, sans donner d'informations supplémentaires. Par exemple, il serait souhaitable de savoir, dans le cas où un système de tâches n'est pas ordonnançable, si une petite modification peut rendre le système ordonnançable, ou si, au contraire, il faut revoir le système en entier.
L'objectif de l'analyse de sensibilité est d'obtenir des informations sur les conséquences de la modification des paramètres des tâches d'un système sur l'ordonnançabilité de ce même système. Ceci a deux avantages majeurs :

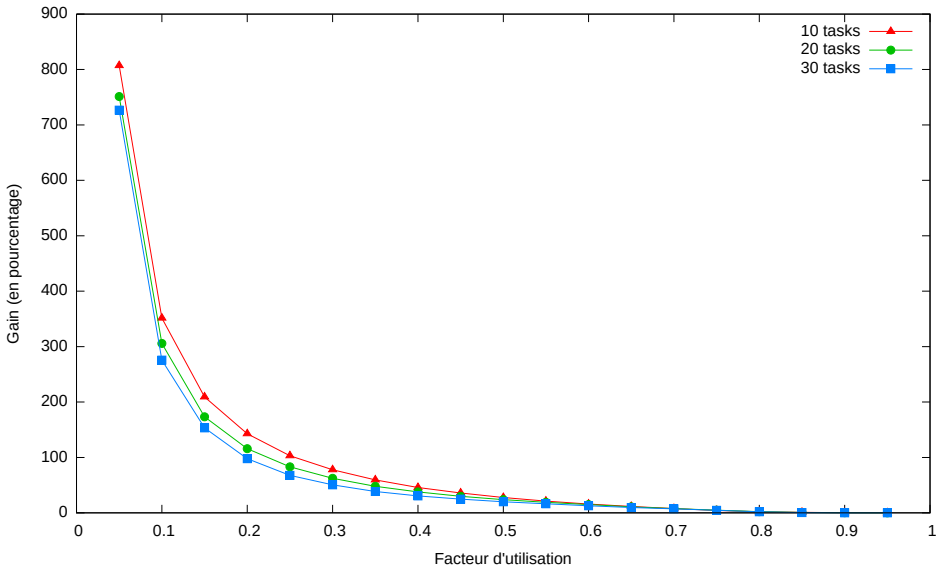


FIGURE 4.4 – Evolution du gain en fonction du nombre de tâches

– dans le cadre de la réalisation d'un nouveau système, les paramètres des tâches peuvent souffrir d'une certaine incertitude, et l'analyse de sensibilité permet donc d'évaluer la marge de manœuvre disponible ;
– dans le cadre d'un système déjà existant, l'analyse de sensibilité peut être utilisée pour connaître la marge de manœuvre disponible dans le cas d'ajout de fonctionnalités par exemple.

Dans les deux cas, l'analyse de sensibilité permet, en partant d'un système de tâches connu, de déterminer la marge de manœuvre disponible quant à la modification des temps d'exécution et / ou des périodes des tâches sans modifier l'ordonnançabilité du système.

De nombreux travaux ont été réalisés à ce propos, principalement au niveau de l'analyse du pire temps d'exécution, de manière à déterminer la valeur maximale que puisse prendre les temps d'exécution tout en s'assurant que toutes les échéances soient respectées [LSD89, KRP+93, KAS93, PDB97, Ves94]. Ces travaux ont ensuite été étendus afin de prendre en compte des systèmes temps-réel plus complexes, comme dans [RHE08], des analyses multidimensionnelles [RHE06] ou des modifications des caractéristiques temporelles des tâches (temps d'exécution, échéance, période) [BDNB08, BB09, BRC09] pour ne citer que quelques travaux parmi les plus récents à ce sujet.

III.2. Méthode de Bini

L'analyse de sensibilité a beaucoup été étudiée dans le passé, et nous avons même déjà utilisé, de manière indirecte, cette analyse au travers de la notion de facteur d'échelle critique [LSD89]. Nous nous intéressons aux travaux de Bini *et al.* [BDNB08], qui ont développé deux méthodes,

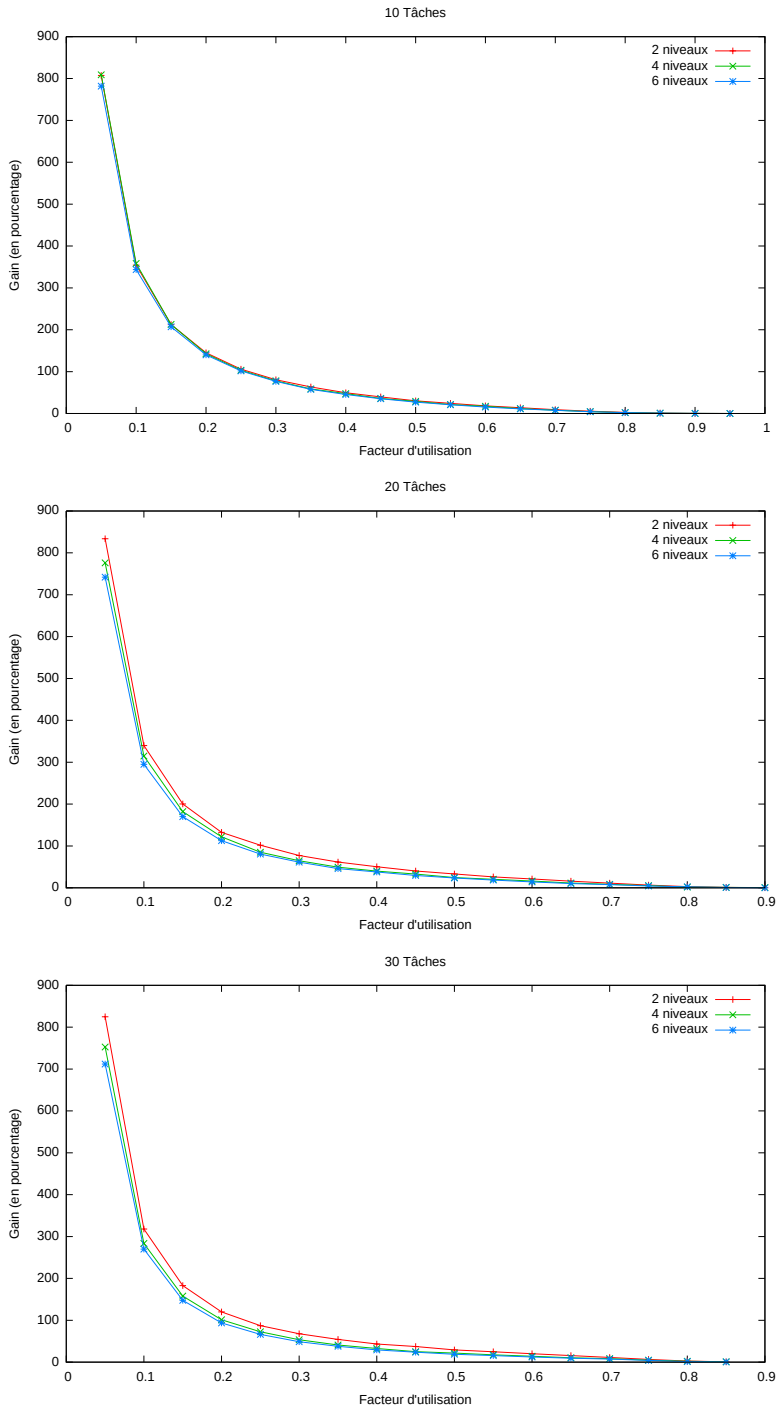


FIGURE 4.5 – Influence du nombre de niveaux de criticité

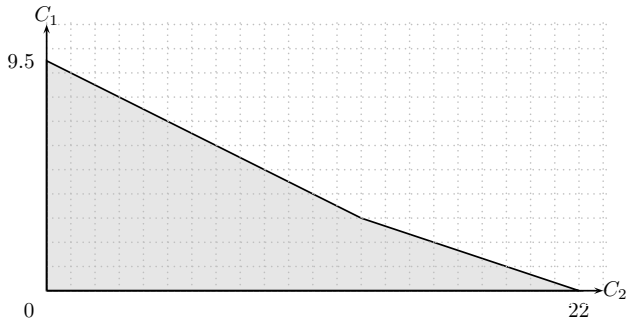


FIGURE 4.6 – Exemple de représentation du $\mathbb{C}$-espace pour un système composé de 2 tâches, avec $T_1 = D_1 = 9.5$ et $T_2 = D_2 = 22$

dont une qui généralise la notion de facteur d'échelle critique. Les deux méthodes sont :
- Une analyse du $\mathbb{C}$-espace, c'est-à-dire l'étude de l'influence de la modification des temps d'exécution des tâches sur l'ordonnançabilité d'un système ;
- une analyse du f-espace, c'est-à-dire l'étude de l'influence de la modification des périodes des tâches sur l'ordonnançabilité d'un système.

Ces méthodes permettent de représenter graphiquement ces espaces, ou domaine de validité. La figure 4.6 est un exemple de $\mathbb{C}$-espace pour un système composé de deux tâches.
Dans la suite de cette thèse, nous nous intéresserons particulièrement à l'analyse de sensibilité dans le $\mathbb{C}$-espace développée par Bini *et al.*, qui permet de choisir la direction dans laquelle nous voulons réaliser l'analyse de sensibilité, c'est-à-dire de choisir quel sous-ensemble de tâches nous souhaitons étudier.
Le point de départ de la méthode est la condition nécessaire et suffisante d'ordonnançabilité pour un système de tâches à échéance contrainte :

$$\max_{1 \leq i \leq n} \min_{t \in S_i} \sum_{j=1}^{i} C_j \left\lceil \frac{t}{T_j} \right\rceil \leq t \tag{4.32}$$

ou, sous forme vectorielle :

$$\max_{1 \leq i \leq n} \min_{t \in S_i} \mathbb{C}_i n_i(t) \leq t \tag{4.33}$$

où $\mathbb{C}_i$ est un vecteur des i tâches de plus forte priorité $\mathbb{C}_i = (C_1, C_2, \ldots, C_i)$, et $n_i(t) = \left(\left\lceil \frac{t}{T_1} \right\rceil, \left\lceil \frac{t}{T_2} \right\rceil, \ldots, \left\lceil \frac{t}{T_{i-1}} \right\rceil, 1 \right)$.
En remplaçant $\mathbb{C}_i$ par $\mathbb{C}_i + \lambda d_i$ dans l'équation 4.33, nous obtenons (la preuve complète peut être trouvée dans [BDNB08]) :

$$\lambda = \min_{i=1,\ldots,n} \max_{t \in sched(P_i)} \frac{t - n_i(t)\mathbb{C}_i}{n_i(t) d_i} \tag{4.34}$$

où λ est le facteur d'échelle et $sched(P_i)$ est un sous-ensemble de S_i.
Le vecteur d_i correspond à la direction étudiée. Si nous voulons réaliser une analyse de sensibilité sur τ_k seulement, alors d_i doit être égale à $(\underbrace{(0,\ldots,0,\overbrace{1}^{k^{\text{ème}}\text{élément}},0,\ldots,0)}_{i \text{ éléments}})$.
Si nous voulons réaliser une analyse de sensibilité sur le système entier, alors d_i doit être égale à $\mathbb{C}_i$. L'analyse correspondante conduit alors à définir le facteur d'échelle critique du système.
L'analyse de sensibilité dans le $\mathbb{C}$-espace est une généralisation de l'analyse de sensibilité introduite par Lehoczky dans [Leh90] dans le sens où la détermination du facteur d'échelle critique d'une seule tâche ou du système entier est un cas particulier de la méthode développée par Bini *et al.*.

III.3. Adaptation au modèle à criticité multiple

Nous avons adapter l'analyse de sensibilité de Bini *et al.*, initialement développée pour des systèmes de tâches classiques [BDNB08], aux systèmes de tâches à criticité multiple. Nous nous sommes seulement focalisés sur l'analyse de sensibilité dans le $\mathbb{C}$-espace puisque le modèle à criticité multiple se distingue du modèle classique de tâches sporadiques en considérant un ensemble de temps d'exécution pour chaque tâche.
Nous avons étendu l'analyse de sensibilité dans le $\mathbb{C}$-espace en analysant les tâches d'un même niveau de criticité. Au lieu d'avoir un seul λ dans la direction étudié d, nous définissons un λ_ℓ par niveau de criticité ℓ.

$$\lambda_\ell \stackrel{\text{def}}{=} \min_{\substack{i=1,\ldots,n \\ L_i=\ell}} \max_{t \in \text{sched}(P_i)} \frac{t - n_i(t)\mathbb{C}_i(\ell)}{n_i(t)d_i} \tag{4.35}$$

Lors de cette opération, nous devons porter une attention particulière sur les temps d'exécution C_i modifiés. En effet, les modifications peuvent rompre l'hypothèse de base des systèmes à criticité multiple exprimée par l'équation 4.2. En pratique, un tel problème peut facile être résolu en ajoutant l'équation 4.3 comme contrainte à la méthode de Bini *et al.*. Plus précisément, il est nécessaire de normaliser les temps d'exécution de chaque tâche de sorte que l'hypothèse de départ du modèle soit respectée (c'est-à-dire l'équation 4.2).
Pour cela, à chaque fois que l'équation 4.2 est violée, c'est-à-dire :

$$\exists \ell, C_i(\ell) > C_i(\ell+1) \tag{4.36}$$

Alors, nous assignons la valeur de C_i au niveau de criticité $\ell+1$ au C_i au niveau de criticité ℓ :

$$C_i(\ell) \leftarrow C_i(\ell+1) \tag{4.37}$$

τ_i	T_i	D_i	L_i	$C_i(1)$	$C_i(2)$
τ_1	137	65	1	9	29
τ_2	286	139	2	86	86
τ_3	248	168	1	32	160

TABLE 4.5 – Exemple de système de tâches à criticité multiple

III.4. Exemple

Après cette étape de normalisation, l'analyse de sensibilité de Bini *et al.* peut facilement s'appliquer. Etudions l'exemple simple du système de tâches à criticité multiple dont les caractéristiques sont données table 4.5.
Arrêtons-nous sur la tâche τ_2 sur laquelle nous allons réaliser l'analyse de sensibilité. Bini *et al.* ont montré dans [BDNB08] que lorsque l'analyse de sensibilité est réalisée sur une seule tâche, l'équation 4.35 [6] peut être réécrite en :

$$\delta C_k^{\max} = \min_{i=k,\ldots,n} \max_{t \in \text{sched}(\mathrm{P_i})} \frac{t - n_i(t)\mathbb{C}_i}{\left\lceil \frac{t}{T_k} \right\rceil} \tag{4.38}$$

Pour appliquer l'analyse de sensibilité, nous devons déterminer l'ensemble des points d'ordonnancement. Dans [BB04b], Bini utilise une méthode récursive pour les déterminer :

$$\begin{cases} \text{sched}(\mathrm{P_i}) & \stackrel{\text{def}}{=} & \mathcal{P}_{i-1}(D_i) \\ \mathcal{P}_0(t) & \stackrel{\text{def}}{=} & \{t\} \\ \mathcal{P}_i(t) & \stackrel{\text{def}}{=} & \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \bigcup \mathcal{P}_{i-1}(t) \end{cases} \tag{4.39}$$

Appliquer l'équation 4.39 à τ_2 et τ_3 pour avoir leurs points d'ordonnancement nous donne les ensembles suivants :

$$\text{sched}(\mathrm{P_2}) = \{T_1, D_2\} \tag{4.40}$$
$$\text{sched}(\mathrm{P_3}) = \{T_1, D_3\} \tag{4.41}$$

Ainsi, nous pouvons maintenant déterminer le facteur d'échelle critique pour les tâches τ_2 et τ_3 (la table 4.6 contient une trace des calculs) :

$$\delta C_2 = \max(22, -5) = 22 \tag{4.42}$$
$$\delta C_3 = \max(10, 32) = 32 \tag{4.43}$$

6. Nous n'utilisons pas les notations utilisées par Bini *et. al* ΔC_k^{max} pour éviter la possible confusion avec le facteur d'échelle critique. Nous utilisons la notation δC_k^{max} à la place.

τ_2		τ_3	
t	δC_2	t	δC_3
137	22	137	10
139	-5	168	32

TABLE 4.6 – Trace des δC_i

τ_i	T_i	D_i	L_i	$C_i(1)$	$C_i(2)$
τ_1	137	65	1	9	29
τ_2	286	139	2	118	108
τ_3	248	168	1	32	160

TABLE 4.7 – Analyse de sensibilité avant l'étape de normalisation

Ayant ces δC_i, nous pouvons maintenant déterminer le facteur d'échelle critique de chaque niveau de criticité pour la tâche τ_2 :

$$\begin{aligned} \delta C_2^{\max}(1) &= \min_{i=1,\dots,n \wedge L_i=1}(\delta C_i) \\ &= \min(\{\delta C_3\}) \qquad (4.44) \\ \delta C_2^{\max}(2) &= \min_{i=1,\dots,n \wedge L_i=2}(\delta C_i) \\ &= \min(\{\delta C_2\}) \qquad (4.45) \end{aligned}$$

Si nous appliquons les modifications au système de tâches, nous obtenons le système dont les caractéristiques sont données table 4.7. Nous pouvons donc constater que l'hypothèse de départ du modèle de tâches à criticité multiple n'est pas respectée (équation 4.2) pour la tâche τ_2 puisque $C_2(1) > C_2(2)$. Aussi, nous devons réaliser l'étape de normalisation telle qu'elle est décrite dans la section précédente.

La normalisation nous conduit au système décrit table 4.8. La figure 4.7 montre le $\mathbb{C}$-espace à criticité multiple pour la tâche τ_2, c'est-à-dire l'ensemble des valeurs possibles pour $C_2(1)$ et $C_2(2)$ pour que soit vérifiée l'équation 4.2.

τ_i	T_i	D_i	L_i	$C_i(1)$	$C_i(2)$
τ_1	137	65	1	9	29
τ_2	286	139	2	108	108
τ_3	248	168	1	32	160

TABLE 4.8 – Analyse de sensibilité après l'étape de normalisation

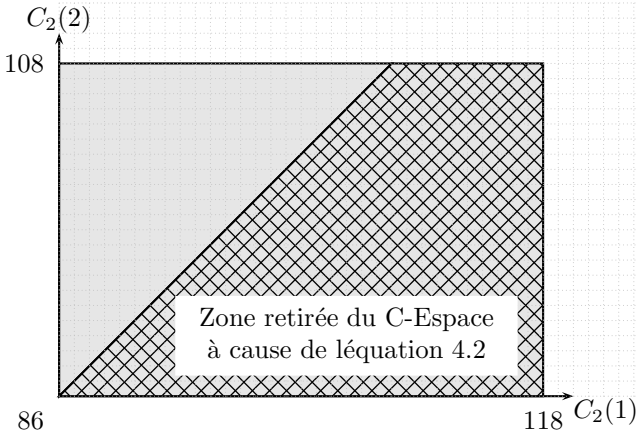


FIGURE 4.7 – $\mathbb{C}$-espace à criticité multiple pour la tâche τ_2

IV. Gigue sur activation

IV.1. Gigue sur activation classique

Dans [ABR+93], Audsley *et al.* ont introduit la notion de gigue sur activation lors de l'assignation des priorités à des tâches. Le but de la gigue sur activation est de modéliser un délai entre l'arrivée d'une tâche et son activation (par exemple, pour modéliser les délais introduits par le noyau temps-réel ou le délai introduit par l'attente de communications entrantes). Les gigues sur activation ont besoin d'être prises en compte dès lors que l'hypothèse selon laquelle les tâches sont activées dès leur arrivée n'est plus vérifiée. Ainsi, prendre en compte la gigue sur activation permet d'avoir des tâches qui sont retardées après leur arrivée dans le système. La gigue sur activation de la tâche τ_i est notée J_i.
Nous allons voir qu'introduire la notion de gigue sur activation dans des systèmes de tâches à criticité multiple est aussi simple qu'ajouter les gigues sur activation dans le modèle défini par Liu et Layland dans [LL73]. La prise en compte de la gigue sur activation lors de la détermination du pire temps de réponse se fait de la manière suivante :

$$Tr_i = \sum_{j=1}^{i} \left\lceil \frac{Tr_i + J_j}{T_j} \right\rceil C_j(L_i) \tag{4.46}$$

IV.2. Gigue à criticité multiple

Dans cette section, nous ne faisons aucune hypothèse quant aux liens éventuels qui pourraient exister entre les gigues sur activation et les tâches à criticité multiple. Plus précisément, les gigues sur activation peuvent modéliser les délais dûs au noyau temps-réel qui peuvent être évalués en utilisant des méthodes différentes pour estimer le temps d'exécution correspondant.

Aussi, nous pouvons considérer que la gigue sur activation n'est plus une constante, mais dépend du niveau de criticité des tâches. Par exemple, pour des tâches faiblement critiques, la gigue sur activation peut être négligée tandis que pour des tâches hautement critiques, il faudra tenir compte d'une gigue non nulle, soumise aux mêmes contraintes que les temps d'exécution des tâches, c'est-à-dire :

$$J_i(\ell) \leq J_i(\ell+1), \forall \ell \tag{4.47}$$

Dans ce cas, prendre en compte cette gigue à criticité multiple est aussi simple que la prise en compte d'une gigue sur activation classique. La seule différence est que le niveau de criticité de la tâche doit être pris en compte. En conséquence, l'équation 4.46 devient :

$$Tr_i = \sum_{j=1}^{i} \left\lceil \frac{Tr_i + J_j(L_i)}{T_j} \right\rceil C_j(L_i) \tag{4.48}$$

où $J_j(\ell)$ représente la gigue sur activation de τ_j au niveau de criticité ℓ.

V. Conclusion

Dans ce chapitre, nous avons étudié le modèle de tâches à criticité multiple introduit dans [Ves07] et [BV08]. Un tel modèle de tâches représente une avancée potentiellement significative dans la modélisation des systèmes temps réel. Nous avons tout d'abord prouvé formellement que l'algorithme d'Audsley est optimal dans la catégorie des algorithmes à priorité fixe pour ordonnancer des systèmes de tâches indépendantes à échéance contrainte, et qu'en conséquence, la règle de sélection introduite par Vestal n'est pas utile d'un point de vue de l'optimalité.
De plus, nous avons réalisé deux types d'analyse de sensibilité : nous avons tout d'abord montré que l'algorithme de Vestal peut être étendu pour déterminer la vitesse processeur minimum pour qu'un système de tâches soit ordonnançable. C'est dans ce but qu'est utilisée la notion de facteur d'échelle critique introduite par Lehoczky, en tant que critère de sélection. Nos expérimentations ont montré que le gain de l'algorithme de Vestal par rapport à l'algorithme d'Audsley est surtout sensible pour des systèmes de tâches à faible facteur d'utilisation.
Nous avons complété cette analyse de sensibilité par l'adaptation des travaux de Bini dans [BDNB08] concernant l'analyse de sensibilité dans le $\mathbb{C}$-espace.
Enfin, nous avons introduit les gigues sur activation dans le modèle de tâches à criticité multiple, en montrant notamment que les précédents travaux d'Audsley [ABR+93] peuvent être facilement adaptés à ce modèle. Pour cela, nous avons introduit la notion de gigue à criticité multiple, c'est-à-dire introduit des délais qui seront évalués en fonction du niveau de criticité considéré.

Publications. Nos travaux ont fait l'objet de deux publications à des conférences avec comité de sélection :

– l'une de nos publications a été acceptée à la conférence internationale RTNS'09 [DRRG09] qui s'est déroulée à Paris en octobre 2009. Cette publication a été récompensée par le titre de meilleur article d'étudiant.
– notre autre publication a été acceptée à la conférence internationale PMS'10 [7] [DGRR10], qui s'est déroulée à Tours en avril 2010.

Enfin, l'article présenté à RTNS'09, a été sélectionné en vue d'une éventuelle publication dans RTS Journal [8]. Il a donc été étendu, et est actuellement en deuxième relecture.

7. *Project Management and Scheduling*
8. *Real Time System Journal*

Chapitre

5 Ordonnancement semi-partitionné avec migrations restreintes

Sommaire

Résumé

Les algorithmes utilisant une stratégie semi-partionnée pour ordonnancer un système de tâches sur une architecture multiprocesseur repose sur le même principe que ceux utilisant une stratégie par partitionnement : ils affectent chaque tâche à un processeur donné. La différence réside dans le fait que certaines tâches, celles qui ne peuvent pas être ordonnancées sur un processeur, peuvent migrer d'un processeur à l'autre. Ce chapitre présente notre contribution au travers la présentation de 2 algorithmes.

I. Introduction

Même si l'algorithme EDF n'est plus optimal pour les architectures multiprocesseurs, de nombreux algorithmes alternatifs basés sur EDF ont été développés à cause de son optimalité pour les architectures monoprocesseurs. L'objectif premier des concepteurs de systèmes temps-réel est de pouvoir traiter des systèmes à la charge grandissante tout en s'assurant du respect des échéances. Malheureusement, la plupart des algorithmes développés dans la littérature souffrent d'un écart entre l'ordonnançabilité théorique et le surcoût induit par l'algorithme lors de l'exécution du système : ordonnancer des systèmes fortement chargés conduit à des calculs complexes.

Depuis plusieurs années, l'ordonnancement de systèmes temps-réel sur des architectures multiprocesseurs a reçu une attention considérable [BB06, GBF02]. Pour de tels systèmes, la plupart des résultats obtenus sont soit des algorithmes utilisant une *stratégie globale* (cf. chapitre 1 section IV.5), soit des algorithmes utilisant une *stratégie par partitionnement* (cf. chapitre 1 section IV.4). Malheureusement, les algorithmes utilisant une stratégie globale induisent souvent un surcoût processeur non négligeable (surcoût dû au noyau, principalement pour la gestion de la préemption et de la migration des tâches), tandis que les algorithmes utilisant une stratégie par partitionnement nécessiteraient parfois de partitionner une tâche (c'est-à-dire permettre son exécution sur plusieurs processeurs) afin de pouvoir ordonnancer un système de tâches.

C'est pourquoi de récents travaux ont mis à jour une nouvelle catégorie d'algorithmes [AB08, AT06, KY07, KY08a, KY09] : les algorithmes utilisant une *stratégie semi-partitionnée*, avec pour objectifs de :

– diminuer les surcoûts à l'exécution, notamment ceux dûs aux migrations des tâches ;
– améliorer l'ordonnançabilité et le facteur d'utilisation du système.

Le principe des algorithmes utilisant une stratégie semi-partitionnée est d'assigner la plupart des tâches à un processeur donné (de la même manière qu'avec une stratégie par partitionnement), ce qui permet de réduire le surcoût à l'exécution, tandis que quelques tâches vont être autorisées à migrer d'un processeur à un autre dans le but d'augmenter l'ordonnançabilité du système.

Des résultats pertinents en terme d'ordonnançabilité et de facteur d'utilisation ont été obtenus en utilisant cette technique. Toutefois, la plupart de ces techniques était basée sur une stratégie utilisant le partitionnement des instances [AB08, ABB08, KY07, KY08b, KY09], c'est-à-dire que les instances d'une tâche peuvent être découpées pour être réparties sur différents processeurs (en d'autres termes, les instances d'une tâches peuvent migrer d'un processeur à un autre en cours d'exécution). Un exemple de ce type d'algorithme est donné section I.3.

Nous avons utilisé une approche différente : nous avons développé un algorithme autorisant la migration des tâches, mais interdisant la migration des instances. Une instance qui commence son exécution sur un processeur ne pourra donc s'exécuter que sur ce processeur. Par contre, deux instances successives pourront s'exécuter sur deux processeurs différents.

I.1. Principe des algorithmes semi-partitionnés

Les algorithmes que nous allons présenter reposent sur le même principe : les tâches sont d'abord réparties sur les différents processeurs en utilisant l'algorithme *First Fit Decreasing* (ou *FFD*) [Bak05a] qui sera décrit dans la section suivante ; si une tâche ne peut être affectée à un seul processeur, alors au lieu de déclarer le système comme étant non ordonnançable par FFD, nous allons tenter de répartir la tâche sur plusieurs processeurs. Ainsi donc, le système sera composé de 2 types de tâches :

- des tâches *non migrantes*, qui seront assignées à un processeur particulier et qui s'exécuteront sans migrations, c'est-à-dire, toujours sur le même processeur ;
- des tâches *migrantes*, qui ne seront non plus assignées à un processeur particulier mais à un ensemble de processeurs. Le schéma de migration dépendra alors de l'algorithme utilisé.

Dans les sections suivantes, nous allons tout d'abord décrire l'algorithme FFD (section I.2), qui est une brique essentielle des algorithmes que nous allons étudier par la suite, l'algorithme développé par Shinpei Kato (section I.3), afin de pouvoir comparer nos travaux à des travaux existants, puis le modèle que nous allons utiliser dans le cadre de notre algorithme (section I.4).

I.2. First Fit Decreasing

L'algorithme First Fit Decreasing [Bak05a] repose sur le principe suivant :

- les tâches sont triées par facteur d'utilisation décroissant ;
- chaque tâche τ_i est placée sur le premier processeur P_j pouvant l'accueillir. Pour le vérifier, nous vérifions l'ordonnançabilité du processeur P_j en utilisant l'algorithme EDF (le choix d'EDF est parfaitement justifié puisque, dans ce cas, pour vérifier l'ordonnançabilité des tâches présentes sur P_j, nous ne considérons que P_j, c'est-à-dire que nous nous sommes ramenés à l'étude d'un système monoprocesseur, système pour lequel l'algorithme EDF est optimal).

Si toutes les tâches peuvent être affectées à un processeur, alors le système est ordonnançable. Dans le cas contraire, il sera non ordonnançable par FFD / EDF.

I.3. Algorithme de Shinpei Kato

Le principe de l'algorithme de Shinpei Kato est, lorsqu'une tâche ne peut être placée entièrement sur un processeur en utilisant FFD, de découper l'exécution de la tâche, de sorte que chaque instance de cette tâche commence son exécution sur un processeur et qu'elle la termine sur un autre processeur, en s'exécutant éventuellement sur des processeurs intermédiaires si nécessaire, et ceci en suivant toujours le même schéma de migration (cf. figure 5.1).

La méthode de découpe choisie par Shinpei Kato est une méthode basée sur le remplissage processeur : la taille du morceau qui s'exécutera sur un processeur sera aussi grande que possible, et sera donc limitée :

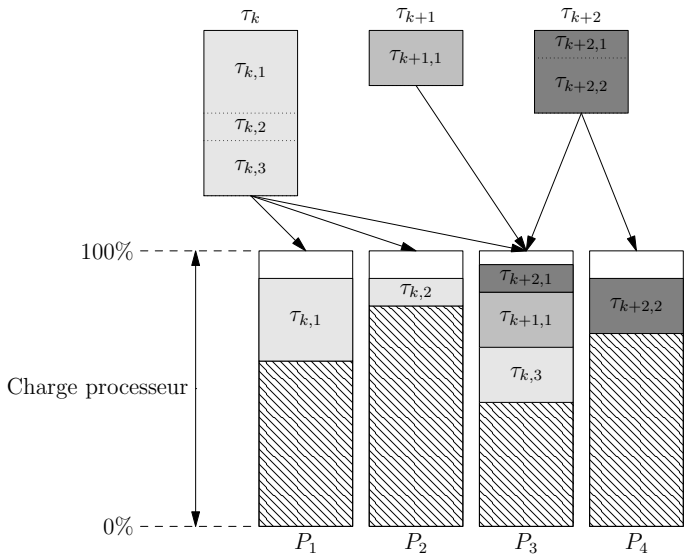


FIGURE 5.1 – Algorithme de Shinpei Kato

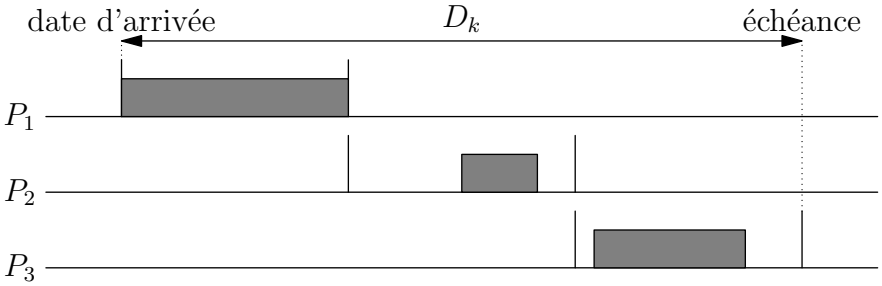


FIGURE 5.2 – Définition des fenêtres d'exécution

– soit par la *capacité du processeur*. Dans ce cas, il faudra au moins un processeur supplémentaire pour exécuter la tâche ;
– soit par le *facteur d'utilisation restant à exécuter*. Dans ce cas, le processeur sera celui sur lequel la tâche terminera son exécution.

Les détails de l'algorithme de Shinpei Kato sont donnés algorithmes 2 et 3.

A cause de cette stratégie de découpage des instances des tâches, il est nécessaire d'avoir un mécanisme qui assure qu'une instance ne peut pas être exécutée en parallèle sur deux processeurs ou plus. Un tel mécanisme a été mis en place par l'introduction de *fenêtres d'exécution* (cf. figure 5.2) : chaque partition ne peut s'exécuter que dans sa fenêtre d'exécution. Aussi, les exécutions en parallèle sont évitées en s'assurant que les fenêtres d'exécution ne se chevauchent jamais.

Algorithme 2 : Algorithme de partitionnement de Shinpei Kato

Input : m, τ_k
Output :
Function `ShinpeiKato` ($\mathrm{In}\, m, \mathrm{In}\, \tau_k$) **begin**
  $s \leftarrow 2$;
  while $s \leq m$ **do**
    $D_k^{'} = \frac{D_k}{s}$;
    for $x = 1 \,\mathrm{To}\, m$ **do**
      $c_{k,x}^{'} = \texttt{CalcExecTime}(\tau_k, P_x)$;
    Soit $c_{k,z}^{'}$ le $s^{\text{ème}}$ plus grand élément de $\{c_{k,x}^{'} | 1 \leq x \leq m\}$;
    $\mathcal{S}_k = \{P_x | c_{k,x}^{'} \geq c_{k,z}^{'}\}$;
    if $\sum_{P_x \in \mathcal{S}_k} c_{k,x}^{'} < c_k$ **then**
      $s \leftarrow s + 1$;
    else
      $c_{k,z}^{'} = c_{k,z}^{'} - \left(\sum_{P_x \in \mathcal{S}_k} c_{k,x}^{'} - c_k\right)$;
      foreach $P_x \in \mathcal{S}_k$ **do**
        $\mathcal{R}_x = \mathcal{R}_x \cup \{T_k\}$;
      return Success ;
  return Fail ;
end

I.4. Modélisation du système

Dans la suite de ce chapitre, nous considérerons un système de n tâches $\tau = \{\tau_1, \ldots, \tau_n\}$ que nous voulons ordonnancer sur m processeurs identiques. Le $k^{\text{ème}}$ processeur est représenté par P_k. Les tâches sont représentées en utilisant le modèle de Liu et Layland [LL73] décrit dans le chapitre 1 section II.2.c, c'est-à-dire par un triplet $\tau_i = (C_i, D_i, T_i)$, où C_i représente le pire temps d'exécution de τ_i, D_i son échéance et T_i sa période.

Comme nous l'avons dit dans l'introduction, notre algorithme autorise la migration des tâches mais pas la migration des instances. Les instances des tâches seront réparties sur les processeurs en suivant une *stratégie de migration périodique*. Par exemple, si pour la tâche τ_i nous avons défini la stratégie de migration (P_1, P_2, P_1), alors la première instance de τ_i s'exécutera sur le processeur P_1, la seconde sur le processeur P_2, la troisième sur le processeur P_1, et pour la quatrième instance, nous reprenons le schéma de migration depuis le début, c'est-à-dire que la quatrième instance sera placée sur le processeur P_1, la cinquième instance sur le processeur P_2, et ainsi de suite.

Afin de modéliser cette répartition des instances sur plusieurs processeurs, nous avons utilisé le principe suivant :

- chaque tâche dont les instances vont être réparties sur plusieurs processeurs est dupliquée sur ces processeurs ;
- les tâches dupliquées sont modélisées par des *tâches multiframes*[BCGM99].

Algorithme 3 : CalcExecTime

Input : τ_k, P_x
Output : Nombre d'unités de temps de τ_k exécutables sur P_x
Function `CalcExecTime` $(\text{In}\,\tau_k, \text{In}\,P_x)$ **begin**
  $c'_{k,x} = (1 - \sum_{\tau_i \in \mathcal{R}_x} u_i)T_k$;
  $L^* = \mathcal{H}$;
  foreach $T_i \in \mathcal{R}_x \cup \{\tau_k\}$ **do**
    $\alpha = \left\lceil \frac{D'_k - D_i}{T_i} \right\rceil$;
    while $L = \alpha T_i + D'_k < L^*$ **do**
      $c = \frac{L - dbf(\mathcal{R}_x, L)}{\left\lfloor \frac{L - D'_k}{T_k} \right\rfloor + 1}$;
      if $c < c'_{k,x}$ **then**
        $c'_{k,x} = c$;
        $u'_{k,x} = \frac{c'_{k,x}}{T_k}$;
        $L^* = \max\left\{ \frac{\sum_{\tau_i \in \mathcal{R}_x}(T_i - D_i)u_i + (T_k - D'_k)u'_{k,x}}{1 - \sum_{\tau_i \in \mathcal{R}_x} u_i - u'_{k,x}}, \max(D_i | \tau_i \in \mathcal{R}_x) \right\}$;
      $\alpha \leftarrow \alpha + 1$;
  return $c'_{k,x}$;
end

Une tâche multiframe est une tâche qui possède plusieurs temps d'exécution. Ainsi, pour une tâche multiframe $\tau_i = ((C_i, 0, C_i), D_i, T_i)$, $(C_i, 0, C_i)$ définit le *schéma de répartition* des instances de τ_i sur le processeur P_j et signifie que τ_i aura une première instance qui demandera C_i unités de temps pour s'exécuter, la seconde instance n'en demandera pas, et la troisième demandera également C_i unité de temps. Et ce schéma se répète à partir de la quatrième instance.
Lorsque qu'une tâche τ_i sera répartie entre plusieurs processeurs, nous désignerons par τ_i^j la tâche multiframe issue de τ_i et s'exécutant sur le processeur P_j.
Ainsi, par exemple, nous pouvons voir figure 5.3 que la tâche τ_i est dupliquée en deux tâches multiframes $\tau_i^1 = (C_i, 0, C_i)$ et $\tau_i^2 = (0, C_i, 0)$.
Ayant maintenant le modèle que nous allons utiliser, dans les sections suivantes, décrire les algorithmes de répartition des instances et de validation utilisés.

II. Algorithmes de répartition des instances

II.1. Stratégie d'assignation des instances périodiques

Puisque chaque tâche consiste en un nombre infini d'instances, il existe potentiellement un nombre infini de répartitions de ces instances parmi les différents processeurs. Etant donné que

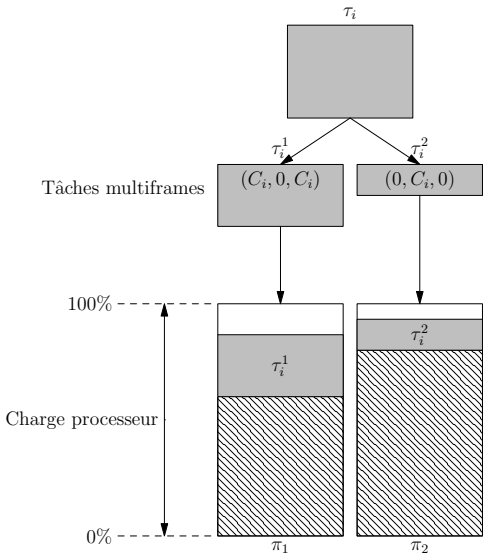


FIGURE 5.3 – Stratégie de répartition

chaque tâche migrante est modélisée par une tâche multiframe sur chaque processeur devant exécuter au moins une instance de la tâche, nous supposons que le nombre de frames obtenues pour chaque tâche migrante est borné par un entier K. Nous supposons, dans le cadre de nos travaux, que K est un paramètre fixé.
Nous désignons par $A[1 \dots n, 1 \dots m]$ un tableau d'entier permettant de connaître la répartition des instances sur les différents processeurs. Le premier index se réfère à une tâche tandis que le second à un processeur. La valeur $A[i, j] \stackrel{\text{def}}{=} x$ indique que x instances de la tâche τ_i s'exécuteront sur le processeur P_j.
Dans les sections suivantes, nous proposons deux algorithmes réalisant l'initialisation de ce tableau et garantissant, pour chaque tâche τ_i, que :

$$\sum_{j=1}^{m} A[i, j] = K \tag{5.1}$$

Ces algorithmes sont :

1. L'algorithme de *répartition uniforme*, qui peut être utilisé lorsque les $A[i, j]$ sont connus *a priori* (ce qui nécessite d'avoir des conditions suffisantes d'ordonnançabilité) ;
2. L'algorithme de *répartition alternatif*, qui peut déterminer les valeurs des coefficients $A[i, j]$ d'une manière gloutonne.

Il est également intéressant de noter que si $K = 1$, alors les migrations des tâches sont interdites et notre méthode se comportera alors comme l'algorithme FFD.

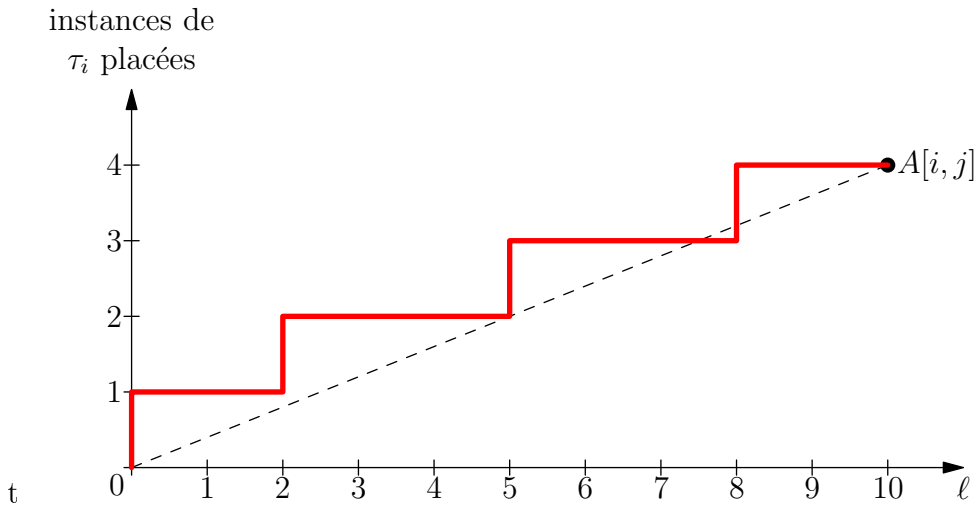


FIGURE 5.4 – Séquence d'assignation des instances

II.2. Stratégie de répartition uniforme

Une *répartition uniforme* de chaque tâche migrante τ_i parmi le sous-ensemble de processeurs P_j qui exécuteront au moins une instance de la tâche τ_i semble être une bonne idée à première vue (mais nous n'avons pas de résultats théoriques le prouvant). Pour cette raison, nous avons introduit le principe de cette stratégie, afin de la tester expérimentalement.
La stratégie de répartition des instances est réalisée selon le principe suivant :

- pour chaque processeur P_k sur lesquels au moins une instance de la tâche τ_i sera exécutée nous déterminons une *séquence d'assignation régulière*, basée sur la fonction en escalier définie par l'équation 5.2 dont une représentation est donnée figure 5.4.

$$f_{i,j}(\ell) \stackrel{\text{def}}{=} \left\lceil \frac{\ell+1}{K} \cdot A[i,j] \right\rceil \tag{5.2}$$

  A chaque saut au niveau de la fonction $f_{i,j}$ correspond une assignation d'une instance de la tâche τ_i sur le processeur P_j.
- si, pour un ℓ donné, plusieurs instances sont assignées à différents processeurs, alors l'ordre d'assignation des instances aux processeurs se fera par index de processeur croissant : pour 2 instances devant être placées sur les processeurs P_j et P_k, si $j < k$ alors la première instance de τ_i est placée sur P_j et la seconde sur P_k.

D'une manière plus formelle, nous pouvons définir la séquence ainsi :

$$\sigma \stackrel{\text{def}}{=} (\sigma_0, \sigma_1, \ldots, \sigma_\ell, \ldots, \sigma_{K-1}) \tag{5.3}$$

$$\sigma_\ell \stackrel{\text{def}}{=} (\sigma_\ell^1, \sigma_\ell^2, \ldots, \sigma_\ell^m) \tag{5.4}$$

$$\sigma_\ell^j \stackrel{\text{def}}{=} \begin{cases} P_j \text{ si } \left\lceil \frac{\ell+1}{K} \cdot A[i,j] \right\rceil - \left\lceil \frac{\ell}{K} \cdot A[i,j] \right\rceil = 1 \\ \emptyset \text{ sinon} \end{cases} \tag{5.5}$$

Algorithme 4 : Algorithme de semi-partitionnement

```
Input : m, τ = {τ_1, ..., τ_n}, K.
Output : True et A si τ est ordonnançable, False sinon.
Function SemiPart (In m, In K, In τ, Out A)
begin
    for (i=1 ... n) do
        Sched ← False;
        /* Nous essayons d'ordonnancer la tâche en utilisant FFD */
        for (j=1 ... m) do
            if τ_i ordonnançable sur π_j then
                τ_i est placée sur π_j;
                Sched ← True;
        if (Sched == False) then
            /*La tâche τ_i est une tâche migrante. Nous utilisons
            une stratégie de répartition des instances sur les processeurs.*/
            if Algo2(τ_i, K, A) == False then
                /*τ_i n'est pas ordonnançable */
                return False;
    return True;
end
```

Exemple. Afin d'illustrer notre propos, nous allons considérer une plateforme constituée de 3 processeurs identiques, P_1, P_2, et P_3. Nous supposons également que le paramètre K est initialisé à 11. Pour une tâche migrante τ_i, nous supposerons que :
- $A[i,1] = 4$
- $A[i,2] = 2$
- $A[i,3] = 5$

c'est-à-dire que 4 instances de τ_i sont assignées au processeur P_1, 2 au processeur P_2 et 5 au processeur P_3. Les séquences d'assignation permettant de déterminer le schéma de migration des tâches sont calculées en utilisant l'équation 5.5 et sont résumées table 5.1.
Par exemple, $\sigma_5 = (\sigma_5^1, \sigma_5^2, \sigma_5^3) = (P_1, P_2, \emptyset)$. Ceci implique donc qu'une instance sera assignée au processeur P_1, puis une autre au processeur P_2, et aucune sur le processeur P_3. Il est à noter que les séquences $\sigma_k^j = \emptyset$ peuvent être ignorées lors de la détermination du schéma de migration. Dans notre exemple, le schéma de migration complet des instances de τ_i est donc :

$$\begin{aligned} \sigma &= (\sigma_0, \sigma_1, \ldots, \sigma_{K-1}) \\ &= (P_1, P_2, P_3, P_1, P_3, P_3, P_1, P_2, P_3, P_1, P_3) \end{aligned}$$

En utilisant cette stratégie, et même si elle réduit considérablement le nombre de migrations comparé à un algorithme autorisant la migration des instances, il est nécessaire de connaître *a priori* le nombre d'instances assignées à chaque processeur. En effet, pour connaître le nombre $A[i,j]$ d'instances de la tâche τ_i qui peuvent être assignées au processeur P_j, nous avons besoin

i	σ_i^1	σ_i^2	σ_i^3
σ_0	1	1	1
σ_1	0	0	0
σ_2	1	0	1
σ_3	0	0	0
σ_4	0	0	1
σ_5	1	1	0
σ_6	0	0	1
σ_7	0	0	0
σ_8	1	0	1
σ_9	0	0	0
σ_{10}	0	0	0
total	4	2	5

TABLE 5.1 – Séquence d'assignation uniforme

d'un test d'ordonnançabilité. Cependant, un tel test nécessite de connaître le schéma de répartition qui nécessite lui-même de connaître la séquence d'assignation des tâches multiframes sur chaque processeur a priori. Ce qui est impossible, puisque nous avons besoin pour cela du nombre d'instances à affecter à chaque processeur. Aussi, nous n'avons d'autre choix ici que de considérer un scénario qui ne prend en considération que le nombre d'instances s'exécutant sur un processeur, c'est-à-dire de considérer un scénario pire cas, qui, bien entendu, introduit du pessimisme.
Dans le but d'illustrer cette impossibilité de connaître le schéma de répartition d'une tâche multiframe sans avoir les séquences d'assignation des tâches multiframes sur les différents processeur, nous allons considérer l'exemple suivant.

Exemple. Considérons la même tâche τ_i que dans l'exemple précédant, mais en supposant maintenant que :
– $A[i,1] = 4$
– $A[i,2] = 1$
– $A[i,3] = 6$
En suivant exactement la même approche que précédemment, nous obtenons le schéma de migration $\sigma = (P_1, P_2, P_3, P_3, P_1, P_3, P_1, P_3, P_3, P_1, P_3)$.
D'un point de vue de l'analyse d'ordonnançabilité, cela signifie que, pour le processeur P_1, la tâche multiframe qui sera considérée est $\tau_i^1 = ((C_i, 0, 0, 0, C_i, 0, C_i, 0, 0, C_i, 0), D_i, T_i)$, tandis que dans l'exemple précédent, nous avions $\tau_i^1 = ((C_i, 0, 0, C_i, 0, 0, C_i, 0, 0, C_i, 0), D_i, T_i)$. Ceci signifie donc que le schéma de répartition de τ_i^1 s'appuie sur la valeur de $A[i,1]$, mais également sur les valeurs de $A[i,2]$ et $A[i,3]$. Ceci nous mène donc à la conclusion que les schémas de répartition ne peuvent pas être déterminés sans connaître tous les $A[i,j]$ a priori.

Ceci n'est pas sans conséquence. En effet, pour déterminer le nombre d'instances qui peuvent être placées sur un processeur, nous avons besoin de connaître le schéma de migration, qui ne sera connu qu'une fois que nous aurons placé les instances de la tâche τ_i sur les processeurs.
Pour surmonter ce problème, deux solutions ont été envisagées et ont été traitées :
- utiliser un test d'ordonnançabilité basé sur un scénario pire cas, ne prenant en compte que le nombre d'instances s'exécutant sur un processeur et non le schéma de migration (cf. section III.1) ;
- utiliser un autre algorithme de répartition n'ayant pas besoin de cette connaissance *a priori*. Il s'agit de la *stratégie de répartition alternative* (cf. section II.3).

II.3. Stratégie de répartition alternative

Cette stratégie de répartition a été mise au point dans le but d'éviter l'inconvénient majeur mis en avant avec le précédent algorithme. Pour cela, nous considérons chaque processeur individuellement et nous utilisons une stratégie de *lecture verticale* des paramètres de la table 5.1 au lieu d'une *lecture horizontale* (voir algorithme 5).
Le principe de l'algorithme est le suivant : nous allons essayer de placer les instances de τ_i sur les différents processeurs. Pour cela, nous énumérons les processeurs un à un, et, à chaque fois, nous allons essayer d'y placer le maximum d'instances. Aussi, lorsqu'un processeur P_j est énuméré, nous initialisons $A[i,j]$ avec le nombre d'instances J_j restant à placer. Ensuite, nous déterminons une tâche multiframe intermédiaire $\tau_i^{'j}$, qui contiendra J_j frames, en utilisant la séquence d'assignation obtenue de la même manière que pour l'algorithme précédent, c'est-à-dire à l'aide de l'équation 5.2. Une fois la tâche $\tau_i^{'j}$ obtenue, nous déterminons la tâche multiframe τ_i^j, contenant K frames à partir de $\tau_i^{'j}$.
L'intérêt d'utiliser une tâche intermédiaire $\tau_i^{'j}$ est de pouvoir générer un schéma de répartition sans se soucier de la prise en compte d'éventuels conflits avec des tâches multiframes qui auraient été placées sur des processeurs précédents (par conflit, nous entendons éviter que deux instances d'une même tâche ne puissent s'exécuter en parallèle sur deux processeurs distincts). Cette prise en compte des conflits se fera au moment de la création de la tâche τ_i^j à partir de $\tau_i^{'j}$.
Ensuite, si cette tâche multiframe τ_i^j est ordonnançable sur P_j alors nous assignons cette tâche au processeur et passons au processeur suivant. Si la tâche multiframe τ_i^j n'est pas ordonnançable, alors nous recommençons le processus sur le même processeur mais en décrémentant $A[i,j]$. Pour tester l'ordonnançabilité de τ_i^j, nous utilisons l'analyse d'ordonnançabilité développée section III.2.

Exemple. Reprenons l'exemple précédent afin d'illustrer le principe de l'algorithme. Au début de celui-ci, $A[i,1]$ est initialisé avec le nombre d'instances de τ_i restant à placer, c'est-à-dire K. Supposons que nous avons testé les valeurs successives de 11 à 5 pour $A[i,1]$ et, qu'à chaque fois, nous avons obtenu une tâche multiframe non ordonnançable. Nous allons donc maintenant tester $A[i,1] = 4$ et vérifier si la tâche multiframe ainsi obtenue est ordonnançable ou non sur

le processeur P_1. Puisque $A[i,1] = 4$, la tâche multiframe intermédiaire qui sera considérée sera $\tau_i'^1 = ((C_i, 0, C_i, 0, 0, C_i, 0, 0, C_i, 0, 0), D_i, T_i)$ (cf. colonne 1 de la table 5.1). Puisque sur le premier processeur P_1, le nombre de frames K et le nombre d'instances restant à placer J_1 sont égaux, alors $\tau_i^1 = \tau_i'^1$. Supposons maintenant que la tâche τ_i^1 est ordonnançable sur P_1. Nous continuons donc le processus sur le deuxième processeur P_2.

Pour déterminer la tâche multiframe τ_i^2 qui sera exécutée sur le processeur P_2, nous déterminons tout d'abord le nombre de frames restant à placer. Ici, $J_2 = K - A[i,1] = 7$. Puis, nous considérons la tâche multiframe $\tau_i'^2$ avec J_2 frames, et initialisons $A[i,2]$ à J_2. Nous allons considérer que nous avons testé les valeurs de 7 à 3 pour $A[i,2]$ sans succès, et que nous allons donc tester $A[i,2] = 2$. Nous déterminons le schéma de répartition des instances de $\tau_i'^2$ en utilisant la séquence d'assignation que nous obtenons par l'équation 5.5 (en prenant bien $K = J_2 = 7$ dans l'équation). Après cette opération, nous déterminons la tâche multiframe τ_i^2 en utilisant τ_i^1 et $\tau_i'^2$ en suivant les règles suivantes :

- Si la $j^{\text{ème}}$ frame de τ_i^1 est non nulle, alors la $j^{\text{ème}}$ de τ_i^2 est égale à 0 ;
- Si la $j^{\text{ème}}$ frame de τ_i^1 est nulle et correspond à la $q^{\text{ème}}$ frame de τ_i^1 qui est nulle, alors la $j^{\text{ème}}$ frame de τ_i^2 est égale à la $q^{\text{ème}}$ frame de $\tau_i'^2$.

L'idée consiste en fait à utiliser les frames de $\tau_i'^2$ pour occuper les *frames laissées libres* par la tâche τ_i^1. Nous désignons par "les frames libres" les frames pour lesquelles aucune instance de τ_i n'est encore assignée (cf. figure 5.5). Ayant maintenant τ_i^2, nous pouvons vérifier son ordonnançabilité sur le processeur P_2.

Ces règles peuvent se généraliser facilement. Supposons que notre but soit de déterminer la tâche multiframe τ_i^k issue de τ_i et qui s'exécutera sur le processeur P_k. Nous devons dans un premier temps déterminer le nombre d'instances restant à placer $J \stackrel{\text{def}}{=} K - \sum_{q=1}^{k-1} A[i,q]$. Ensuite, nous déterminons $\tau_i'^k$ grâce à l'équation 5.5. Enfin, nous calculons τ_i^k en utilisant l'algorithme 6. Il ne reste alors qu'à tester l'ordonnançabilité de τ_i^k sur P_k.

II.4. Algorithme d'ordonnancement

Tout comme pour les algorithmes d'ordonnancement utilisant une stratégie par partitionnement, les ordonnanceurs ont la même politique d'ordonnancement sur chaque processeur, qui est EDF dans notre cas. Le principe de l'algorithme est alors le suivant :

- les tâches sont triées par facteur d'utilisation décroissant ;
- pour chaque tâche considérée individuellement :
  - nous essayons de trouver un processeur sur lequel la tâche peut être placée en utilisant FFD.
  - si un tel processeur n'a pas été trouvé, alors nous utilisons une des stratégies définies dans les sections précédentes, c'est-à-dire soit la stratégie de répartition uniforme (cf. section II.2), soit la stratégie de répartition alternative (cf. section II.3).

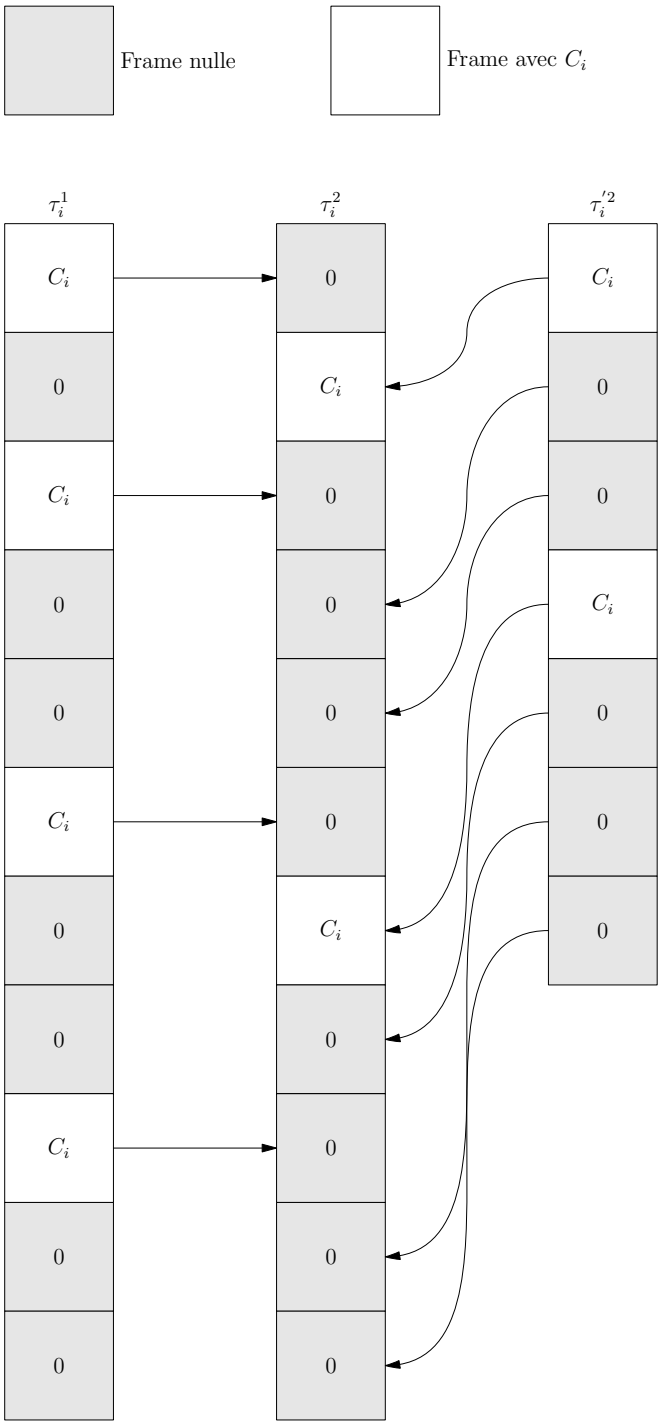


FIGURE 5.5 – Processus de création de τ_i^2

Algorithme 5 : Stratégie d'assignation des instances pour la tâche migrante τ_i

Input : K, tâche τ_i
Output : True et A si τ_i est ordonnançable, False sinon
Function `Algo2` (In τ_i, In K, Out A)
begin
  RemJobs $\leftarrow K$; /* *Instances restantes* */
  for *(*$k = 1 \cdots m$*)* **do**
    for *(*$j =$ RemJobs $\cdots 1$*)* **do**
      Détermination de $\tau_i^{'k}$ en utilisant l'équation 5.5 ;
      $\tau_i^k =$ `ComputeT`$(\tau_i^{'k}, K, (\tau_i^1, \ldots, \tau_i^{(k-1)}))$;
      if *(*τ_i^kest ordonnançable sur π_k*)* **then**
        $A[i,k] \leftarrow j$;
        RemJobs $\leftarrow$ RemJobs $-j$;
        Exit;
    if RemJobs $= 0$ **then return** True ;
  return False
end

Algorithme 6 : Détermination de τ_i^k

Input : K, k, $\tau = (\tau_i^1, \ldots, \tau_i^{(k-1)})$, $\tau_i^{'k}$
Output : Tâche multiframe τ_i^k
Function `Compute` (In $\tau_i^{'k}$, In K, In τ)
begin
  $q \leftarrow 1$;
  for *(*$j = 1 \ldots K$*)* **do**
    Free $\leftarrow$ True;
    $\ell \leftarrow 1$;
    while *(*$\ell < k$ **and** Free*)* **do**
      if $\tau_i^\ell(j) = 0$ **then** $\ell \leftarrow \ell + 1$;
      else Free $\leftarrow$ False;
    if Free **then**
      $\tau_i^k(j) \leftarrow \tau_i^{'k}(q)$;
      $q \leftarrow q + 1$;
    else $\tau_i^k(j) = 0$;
  return τ_i^k ;
end

III. Analyse d'ordonnançabilité

Nos algorithmes utilisent une analyse d'ordonnançabilité pour s'assurer de la validité des systèmes en cours d'ordonnancement. Etant donné que nous modélisons nos tâches migrantes par des tâches multiframes sur chaque processeur sur lesquels s'exécute au moins une instance de ces tâches, chaque processeur peut être tester individuellement, ce qui nous permet d'utiliser des techniques d'analyse d'ordonnançabilité monoprocesseur.
Puisque sur chaque processeur, les tâches sont ordonnancées en utilisant EDF, nous utilisons un test utilisant la *Demand Bound Function* ou *dbf* (cf. chapitre 2 section IV.2). Malheureusement, nous ne pouvons utiliser directement l'approche précédemment décrite, à cause de la présence de tâches multiframes.
Aussi, nous avons développés deux scénarios pour arriver à nos fins :
- un *scénario compact*, qui ne nécessite que la connaissance du nombre d'instances s'exécutant sur un processeur. Nous n'avons donc pas besoin de connaître avec précision la tâche multiframe (c'est-à-dire son schéma de répartition), nous n'avons besoin que de connaître le nombre de frames non nulles ;
- un *scénario avec schéma de répartition*, qui nécessite de connaître exactement le schéma de répartition, c'est-à-dire la tâche multiframe.

Afin de valider ces deux scénarios, nous avons développé des analyses d'ordonnançabilité basées sur une extension de la *dbf* adaptée aux tâches multiframes [BCGM99, BB06].

III.1. Scénario compact

Chaque tâche migrante τ_i est modélisée en utilisant un ensemble de tâches multiframes τ_i^j s'exécutant sur le processeur P_j. Ce scénario considère le cas où chaque tâche multiframe associée à τ_i, à ses temps d'exécution non nuls au début de son schéma de répartition, et le reste étant à 0. τ_i^j est donc modélisée par $\tau_i^j = ((C_i, C_i, \ldots, C_i, 0, \ldots, 0), D_i, Ti)$, où C_i est le temps d'exécution de τ_i.
De plus, ce scénario possède une propriété résumée par le théorème suivant :

Théorème 30. *Soit τ_i une tâche multiframe. Le scénario pire cas est le scénario pour lequel les frames de τ_i sont compactées.*

La démonstration de ce théorème sera faite section III.3.
Afin de définir la *dbf* au temps t, nous avons besoin de déterminer la contribution de chaque tâche dans l'intervalle $[0, t[$. Comme K désigne le nombre de frames des tâches multiframes τ_i^j associées à τ_i, notons ℓ_i^k le nombre de temps d'exécution non nuls de la tâche multiframe τ_i^j. En utilisant le scénario compact, la tâche τ_i^j est donc modélisée par $\tau_i^j = ((\underbrace{C_i, \ldots, C_i}_{\ell_i^j \text{éléments}}, 0, \ldots, 0), D_i, T_i)$.
La difficulté consiste à ne prendre en compte que ℓ_i^k instances de la tâche τ_i^k au plus, puisque au plus ℓ_i^k instances peuvent contribuer à la *dbf* dans l'intervalle $[0, K \cdot T_i]$.
La contribution à la *dbf* au temps t peut être déterminée comme suit (cf. figure 5.6) :

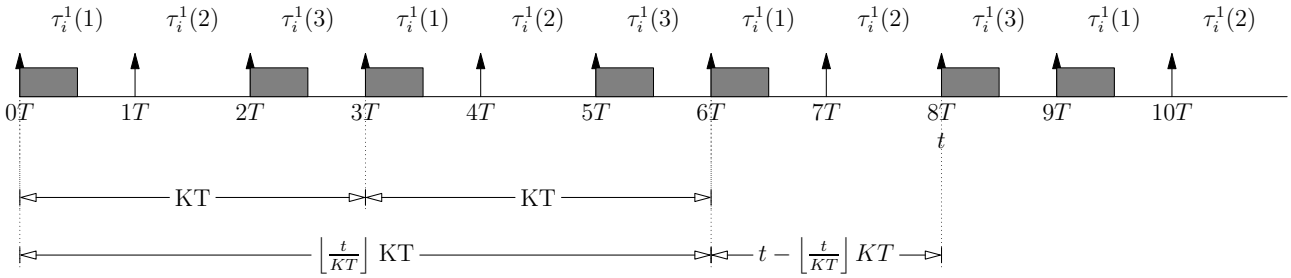


FIGURE 5.6 – Illustration de la tâche multiframe $\tau_i^1 = ((C_i, 0, C_i), T_i, T_i)$.

– Tout d'abord, nous considérons le nombre s d'intervalles de longueur $K \cdot T_i$ au temps t :

$$s \stackrel{\text{def}}{=} \left\lfloor \frac{t}{K \cdot T_i} \right\rfloor \tag{5.6}$$

La contribution dans ces intervalles de la tâche τ_i^k à la dbf vaut :

$$s \cdot \ell_i^k \cdot C_i \tag{5.7}$$

– Ensuite, considérons l'intervalle $[s \cdot K \cdot T_i, t[$, où plusieurs instances peuvent avoir leur échéance avant la date t. En supposant que toutes les instances sont assignées au processeur considéré, le nombre d'instances a est donné par :

$$a \stackrel{\text{def}}{=} \max\left(0, \left\lfloor \frac{(t \bmod K \cdot T_i) - D_i}{T_i} \right\rfloor + 1\right) \tag{5.8}$$

Mais étant donné qu'il n'y a qu'au plus ℓ_i^k instances parmi K qui seront exécutées sur le processeur P_k, la contribution à la dbf pour la tâche τ_i^k dans l'intervalle $[s \cdot K \cdot T_i, t[$ est donc :

$$\min(\ell_i^k, a) \cdot C_i \tag{5.9}$$

En rassemblant toutes ces expressions, nous pouvons définir la dbf de la tâche τ_i^k par :

$$\widehat{dbf}(\tau_i^k, t) \stackrel{\text{def}}{=} s \cdot \ell_i \cdot C_i + \min(\ell_i, a) \cdot C_i \tag{5.10}$$

Analyse d'ordonnançabilité Soit $\tau = \{\tau_1, \ldots, \tau_n\}$ un ensemble de n tâches s'exécutant sur un processeur donné et mélangeant tâches migrantes et non migrantes. Une condition suffisante pour que le système soit ordonnançable est :

$$\sum_{\substack{\tau_i \in \tau \\ \tau_i \text{ non migrante}}} dbf(\tau_i, t) + \sum_{\substack{\tau_i \in \tau \\ \tau_i \text{ migrante}}} \widehat{dbf}(\tau_i, t) \leq t \quad \forall t \tag{5.11}$$

III.2. Scénario avec prise en compte du schéma de répartition

Ce scénario contraste avec le scénario précédent par la prise en compte du schéma de répartition des instances. En effet, nous pouvons supposer que le scénario pire cas que nous avons considéré induit un certain pessimisme, du fait que toutes les instances sont compactées au début du schéma de répartition.
Aussi, nous avons développé un test d'ordonnançabilité, similaire au précédent, mais prenant en compte le schéma de répartition. La différence majeure se situe au niveau de l'évaluation de la *dbf* pour les tâches multiframes. Plus précisément, nous avons remplacé le second terme de l'équation 5.10 par :

$$\max_{c=0}^{K-1} \left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K} \right) \tag{5.12}$$

où $nb_i(t) \stackrel{\text{def}}{=} \left\lfloor \frac{(t \bmod K \cdot T_i) - D_i}{T_i} \right\rfloor + 1$ et correspond au nombre d'instances de τ_i^k dans l'intervalle $[s \cdot K \cdot T_i, t[$. Le principe est de déterminer la demande processeur en considérant que l'instant critique coïncide avec la première instance du schéma, puis avec la seconde, la troisième, etc... jusqu'à la $K^{\text{ème}}$ instance puis nous prenons le maximum des demandes processeurs ainsi calculées. Ce principe nous a été inspiré par les travaux de Rahni dans [RGR08].
Nous obtenons donc, pour le calcul de la *dbf* la fonction suivante :

$$\widehat{\widehat{\mathrm{DBF}}}(\tau_i, t) \stackrel{\text{def}}{=} s \cdot \ell_i \cdot C_i + \max_{c=0}^{K-1} \left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K} \right) \tag{5.13}$$

Analyse d'ordonnançabilité Le test d'ordonnançabilité suit exactement le même principe que le précédent. La seule différence réside dans l'utilisation de $\widehat{\widehat{dbf}}$ au lieu de $\widehat{dbf}$ dans l'équation 5.10.

III.3. Preuve du scénario pire cas

Nous allons maintenant prouver le théorème 30. Pour cela, nous allons introduire les notations utilisées, puis un résultat intermédiaire, résumé par le lemme 4, dont nous aurons besoin dans le cadre de la preuve.
Soit τ_i une tâche multiframe avec le schéma de répartition suivant $\Sigma \stackrel{\text{def}}{=} (\sigma_1, \ldots, \sigma_K)$, c'est-à-dire, $\tau_i = ((\sigma_1 C_i, \ldots, \sigma_K C_i), D_i, T_i)$ avec $\sigma_j \in \{0, 1\}, 1 \leq j \leq K$. Soit ℓ_i^k le nombre de frames non nulles. Soit $\tau_i^{(p)}$ la version compactée de τ_i, c'est-à-dire :

$$\sigma_{i,j} = \begin{cases} 1 \text{ si } j \leq \ell_i \\ 0 \text{ sinon} \end{cases} \tag{5.14}$$

Lemme 4.

$$\widehat{\widehat{dbf}}(\tau_i, t) \leq \widehat{\widehat{dbf}}(\tau_i^{(p)}, t) = \widehat{dbf}(\tau_i^{(p)}, t) \tag{5.15}$$

Démonstration. Par définition de la dbf pour une tâche multiframe avec schéma de répartition (cf. équation 5.13) :

$$\widehat{\widehat{dbf}}(\tau_i, t) = s \cdot \ell_i \cdot C_i + \max_{c=0}^{K-1} \left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K} \right) \tag{5.16}$$

Dans cette équation, le terme max peut être réécrit en :

$$\max_{c=0}^{K-1} \left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K} \right) = \max_{c=0}^{K-1} \left(C_i \sum_{j=c}^{c+nb_i(t)-1} \sigma_{i,j \bmod K} \right) \tag{5.17}$$

Notons également que puisque $\sigma_{i,j} \in \{0, 1\}$ et que nous effectuons une somme de $nb_i(t)$ termes, alors :

$$\sum_{j=c}^{c+nb_i(t)-1} \sigma_{i,j \bmod K} \leq \min(\ell_i, \max(0, nb_i(t))) \tag{5.18}$$

Si maintenant nous prenons en compte le fait que nous n'avons qu'au plus, ℓ_i^k frames non nulles, nous devons prendre le minimum entre ℓ_i^k et $nb_i(t)$:

$$\max_{c=0}^{K-1} \left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K} \right) \leq \max_{c=0}^{K-1} \left(C_i \min(l_i, \max(0, nb_i(t))) \right) \tag{5.19}$$

Puisque le terme de l'opérateur max ne dépend plus de c, nous pouvons supprimer l'opérateur max. Ce qui nous conduit à :

$$\max_{c=0}^{K-1} \left(\sum_{j=c}^{c+nb_i(t)-1} C_{i,j \bmod K} \right) \leq C_i \min(\ell_i, \max(0, nb_i(t))) \tag{5.20}$$

Finalement, nous avons :

$$dbf(\tau_i, t) \leq s \cdot \ell_i \cdot C_i + C_i \min(l_i, \max(0, nb_i(t))) \tag{5.21}$$

C'est-à-dire :

$$\widehat{\widehat{dbf}}(\tau_i, t) \leq \widehat{\widehat{dbf}}(\tau_i^{(p)}, t) \tag{5.22}$$

Ceci prouve le lemme. □

Démonstration du théorème 30. Un scénario $\mathcal{A}$ est dit *pire* qu'un scénario $\mathcal{B}$ si, et seulement si, le fait que le scénario $\mathcal{A}$ soit ordonnançable implique que le scénario $\mathcal{B}$ le soit également. Si nous supposons que le scénario compact est ordonnançable, alors, en utilisant le lemme précédant, ceci implique que le scénario Σ est également ordonnançable. Puisque le scénario Σ représente un scénario quelconque, nous venons de montrer que la validation du scénario compact valide tout autre scénario. Le scénario compact est donc bien le scénario pire cas. Ceci prouve le théorème. □

IV. Résultats expérimentaux

Nous avons réalisé des expérimentations afin de comparer nos algorithmes à FFD et à celui de Shinpei Kato, mais aussi pour voir l'influence que peuvent avoir les paramètres (comme le nombre de frames des tâches multiframes K) sur l'ordonnançabilité des systèmes de tâches.

IV.1. Conditions expérimentales

Nous avons utilisé des systèmes de tâches avec les caractéristiques suivantes :

- le nombre de processeurs m est choisi dans l'ensemble $M \stackrel{\text{def}}{=} \{2, 4, 8, 16, 32, 64\}$. Ce choix n'est pas arbitraire : il reflète la réalité des processeurs multicœurs qui ont généralement un nombre de cœurs qui est une puissance de 2.
- le facteur d'utilisation de chaque système est choisi de sorte que le facteur d'utilisation moyen par processeur varie de 0.5 à 0.95 en utilisant un pas de 0.05.

Afin de pouvoir réaliser des statistiques, nous avons considéré 10 000 systèmes de tâches par configuration (une configuration étant caractérisée par le nombre de processeurs et le facteur d'utilisation moyen par processeur).

Dans le but de pouvoir comparer facilement nos résultats à ceux obtenus par Shinpei Kato, nous avons repris le même protocole de génération des tâches :

- le facteur d'utilisation de chaque tâche est choisi de manière uniforme dans l'intervalle $[0, 1]$, avec la contrainte que la somme des facteurs d'utilisation des tâches doit être égale au facteur d'utilisation du système ;
- la période T_i de chaque tâche τ_i est choisie de manière uniforme dans l'intervalle $[100, 3000]$;
- les tâches sont à échéance sur requête (ie., $D_i = T_i$) ;
- le pire temps d'exécution des tâches est déterminé en fonction de leur facteur d'utilisation et de leur période par $C_i \stackrel{\text{def}}{=} u_i \cdot T_i$.

Il est important de noter que lors de la génération d'un système de tâches, nous n'avons aucun contrôle sur le nombre de tâches que comportera le système.

IV.2. Résultats

La figure 5.7 nous montre l'influence du paramètre K sur le ratio d'ordonnançabilité de l'algorithme utilisant une stratégie de répartition uniforme. Comme nous pouvons le constater, plus la valeur de K est faible, et meilleur est ce ratio. Ce résultat peut sembler surprenant et contre-intuitif à première vue, puisque nous serions en droit de nous attendre à une augmentation de ce ratio lorsque le paramètre K augmente. Une raison possible de ce comportement tient au pessimisme introduit en *compactant* les frames non nulles au début de chaque tâche multiframe. En effet, dans ce cas, la demande processeur à l'instant $t = 0$ est surévaluée pour chaque processeur sur lequel au moins une instance de la tâche s'exécute. Aussi, lorsque $K = 2$, chaque tâche migrante est placée sur au plus 2 processeurs et donc n'induit du pessimisme à l'instant $t = 0$ que sur ces 2 processeurs. Si nous augmentons K, alors le nombre de processeurs sur lequel le pessimisme sera induit sera également plus important.
De la même manière, la figure 5.8 représente l'influence du paramètre K sur le ratio d'ordonnançabilité mais pour l'algorithme utilisant une stratégie de répartition alternative. Nous constatons ici que le résultat contraste avec le scénario pire cas, et que plus K augmente, plus le ratio augmente. L'explication est que, plus K est grand, plus une tâche migrante peut se répartir sur un nombre important de processeurs, diminuant d'autant leur facteur d'utilisation global.
Un point important à noter est que les deux scénarios que nous avons étudiés sont identiques lorsque $K = 2$. En effet, lorsque nous considérons le scénario avec schéma de répartition alternatif, une tâche migrante τ_i sera représentée par 2 tâches multiframes. De manière plus précise, les deux seuls schémas de répartition valides sont alors $(C_i, 0)$ et $(0, C_i)$ (la frame $(0, 0)$ n'est pas autorisée, puisque cela signifierait qu'aucune instance de la tâche ne s'exécute sur le processeur, et la frame (C_i, C_i) induirait que la tâche migrante s'exécuterait toujours sur le même processeur, et serait donc, finalement, non migrante). Or, d'un point de vue de l'ordonnançabilité, chacun de ces points de vue se ramène à l'étude du même schéma, à savoir $(C_i, 0)$, qui correspond à l'unique schéma valide dans le cas du scénario pire cas.
La figure 5.9 compare les performances de nos algorithmes avec l'algorithme classique FFD et l'algorithme de Shinpei Kato. Pour nos algorithmes, nous avons utilisé ceux fournissant les meilleurs performances, c'est-à-dire pour le scénario pire cas, $K = 2$ et pour le scénario avec schéma de répartition alternatif, $K = 20$.
Nous pouvons voir que nos algorithmes dominent toujours FFD. Ceci est tout à fait normal puisque la première étape de nos algorithmes est d'utiliser l'algorithme FFD et de n'autoriser la migration des tâches que si l'algorithme FFD échoue à ordonnancer fiablement une tâche.
Nous constatons également que nos algorithmes fournissent de bons résultats, même s'ils ne sont pas aussi performants que ceux de Shinpei Kato. Nous rappelons toutefois que nos algorithmes n'autorisent que la migration des tâches, et non la migration des instances, contrairement à l'algorithme de Shinpei Kato. De plus, interdisant la migration des instances des tâches, nous réduisons les surcoûts à l'exécution induit par la migration des instances (surcoût dont souffre l'algorithme de Shinpei Kato).

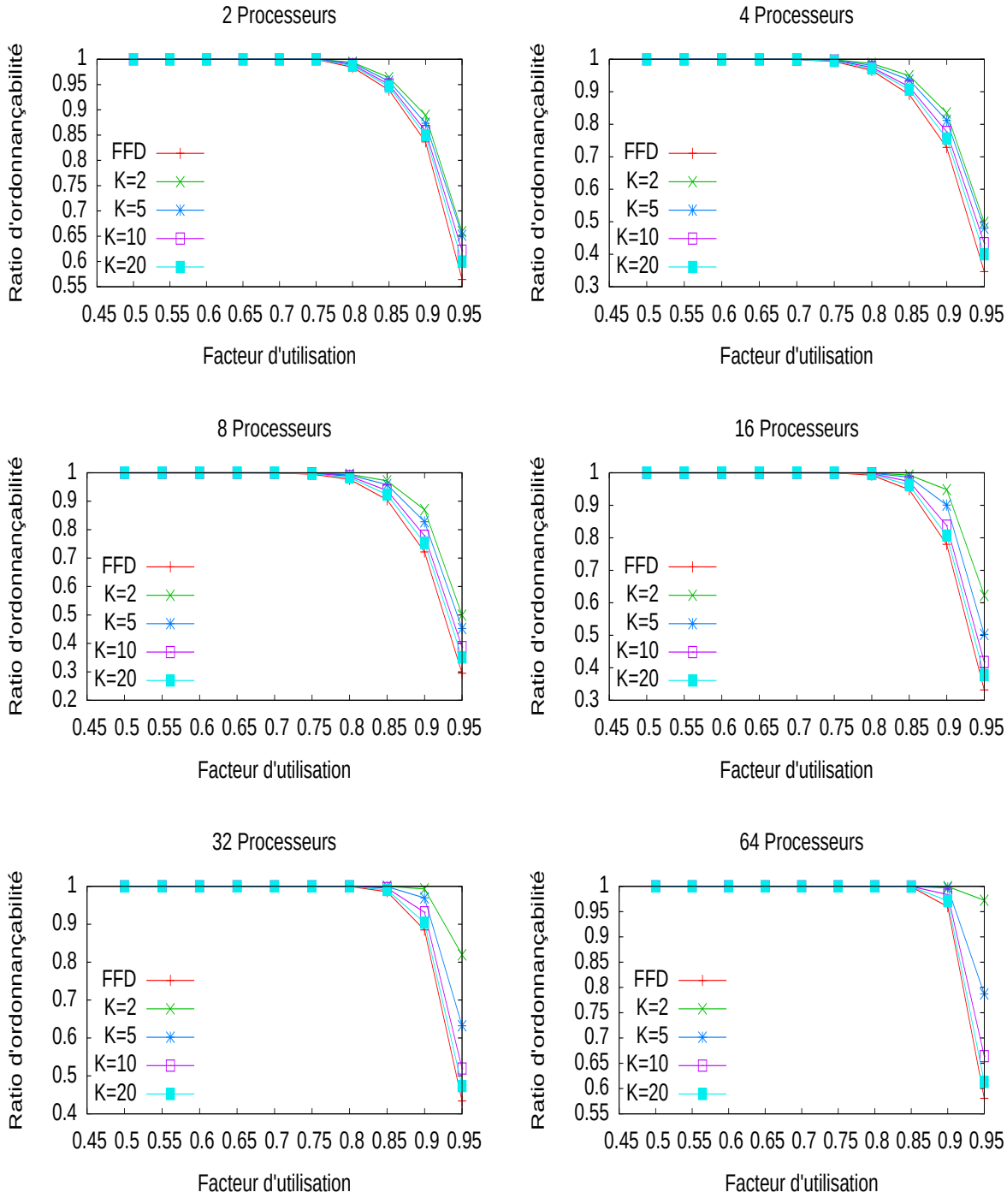


FIGURE 5.7 – Evolution du ratio d'ordonnançabilité en fonction du paramètre K pour les scénarios compacts

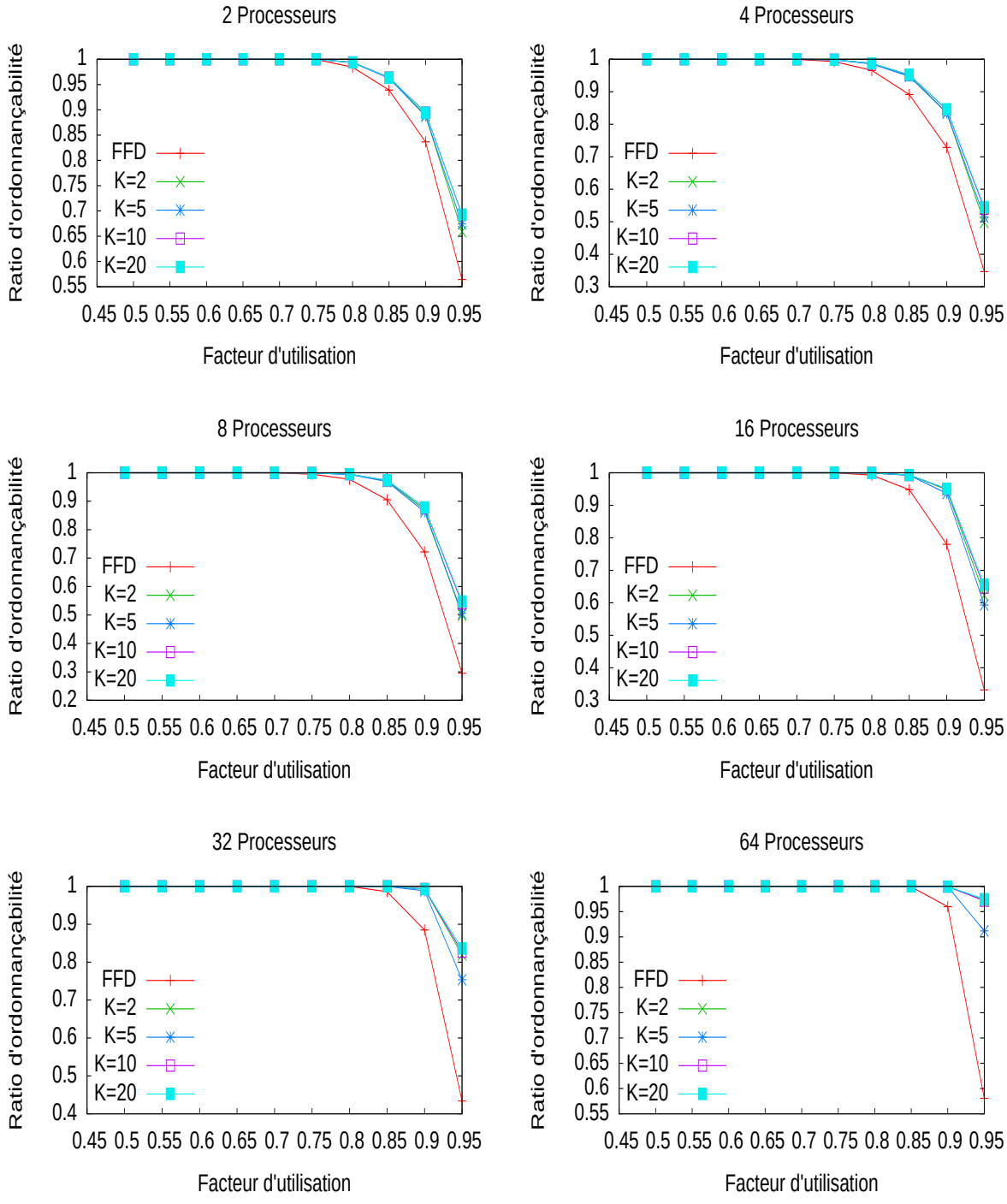


FIGURE 5.8 – Evolution du ratio d'ordonnançabilité en fonction du paramètre K pour les scénarios avec schéma de répartition

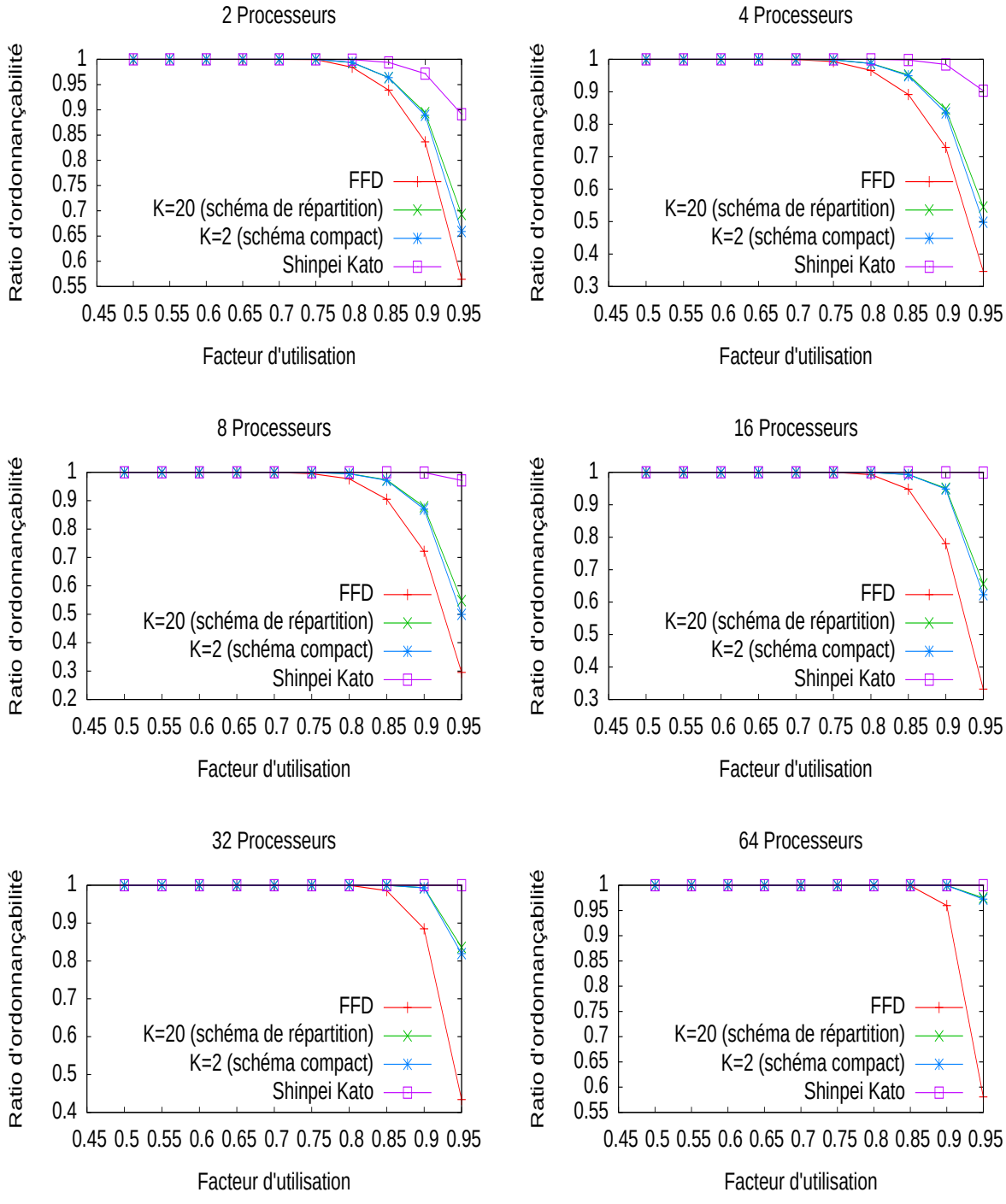


FIGURE 5.9 – Comparaison des algorithmes FFD, compact (K=2), avec schéma de répartition (K=20) et de Shinpei Kato

V. Conclusion

Dans ce chapitre, nous avons étudié le problème de l'ordonnancement d'un système de tâches sur une plateforme multiprocesseur constituée de processeurs identiques. De nouveaux algorithmes, basés sur la notion de semi-partitionnement avec des migrations restreintes ont été présentés, de même que des tests d'ordonnançabilité.
L'efficacité de nos algorithmes a été validée au travers la réalisation de tests expérimentaux, montrant des performances se rapprochant de celles de l'algorithme de Shinpei Kato pour un surcoût à l'exécution moindre, puisque nous n'autorisons que la migration des tâches, et interdisons la migration des instances des tâches.
Parmi les perspectives envisagées, nous pouvons noter :
- relaxer la contrainte sur le paramètre K, et permettre ainsi de définir une valeur pour chaque tâche, au lieu d'utiliser la même valeur pour tout le système ;
- la prise en compte de problème d'optimisation, par exemple, pour minimiser le nombre de migrations.

Publication Les travaux présentés dans ce chapitre ont fait l'objet d'un article [DMYGR10], qui a été soumis à RTNS 2010, et qui est actuellement en relecture.

Conclusion

Nous avons proposé des contributions à l'ordonnancement des systèmes temps réel critiques. L'ordonnancement consiste à déterminer l'ordre d'exécution des tâches informatiques sur les calculateurs. Dans nos travaux, nous nous sommes intéressés aux politiques d'ordonnancement en-ligne, pour lesquelles l'ordre d'exécution des tâches est déterminé sur la base des tâches actives dans le système et sans connaissance a priori des tâches qui arriveront dans le futur. Cette problématique s'étend aux réseaux informatiques pour ordonnancer les messages sur le réseau informatique interconnectant les différents calculateurs du système temps réel. La spécificité de l'ordonnancement temps réel repose sur la périodicité des activations des tâches et sur les contraintes temporelles que doivent respecter les tâches durant leurs exécutions. Le plan du mémoire repose sur une partie état de l'art et trois chapitres de contributions.
Par le travail présenté dans cette thèse, nous avons contribué à l'ordonnancement des systèmes temps réel critique, au travers des aspects suivants :

- tout d'abord, nous nous sommes intéressés à l'étude de systèmes distribués, et plus particulièrement à la minimisation du nombre de processeurs ;
- nous avons ensuite étudié un modèle de tâches alternatif, basé sur le modèle de Liu et Layland mais introduisant la notion de criticité ;
- enfin, nous avons développé des algorithmes semi-partitionné pour des architectures multiprocesseurs tout en restreignant les migrations.

Dans la partie état de l'art, nous avons introduit les systèmes temps réel, ainsi que la problématique de l'ordonnancement. Nous avons présenté des résultats de base qui ont été développés dans la littérature dans le but de répondre à cette problématique. Nous avons également introduit la notion de validation, réalisée durant la conception d'un système temps réel afin de prouver la correction temporelle des applications. Nous avons présenté les méthodes de validation qui ont été utilisées dans le cadre de nos contributions, avec notamment le calcul des pires temps de réponse en priorité fixe monoprocesseur, l'analyse de la demande processeur et l'analyse holistique, prenant en compte les dépendances entre les tâches et les messages, dans

le cas des architectures distribuées.

La première contribution, développée dans le chapitre 3, concerne le placement et ordonnancement simultanés des tâches pour dans un système temps réel distribué, permettant de minimiser le nombre de processeurs nécessaires pour respecter les spécifications temporelles des tâches. Le principe de l'algorithme repose sur l'énumération implicite (méthode de type séparation et évaluation) de tous les ordonnancements possibles, à travers l'exploration d'un arbre de recherche. La technique de validation utilisée est dérivée de l'analyse holistique, qui permet d'obtenir des bornes supérieures des pires temps de réponse des tâches et des messages durant le processus de placement et d'affectation des priorités. Différentes heuristiques permettent d'éliminer les ordonnancements non valides durant la recherche d'une solution optimale.

La seconde contribution concerne les tâches à criticité multiple et fait l'objet du chapitre 4. L'objectif de ce modèle est de prendre en compte la notion de criticité des différentes tâches, comme par exemple dans la norme DO-178B utilisée en aéronautique. En effet, ce standard définit plusieurs niveaux de criticité en fonction des conséquences qu'aurait une défaillance temporelle sur le système, en allant de la tâche critique, dont une défaillance provoquerait le crash d'un avion, à la tâche non critique, dont la défaillance ne serait que peu ou pas notable. L'objectif est donc de pouvoir prendre en compte cette notion de criticité durant la validation du système. Nous avons prouvé que dans ce cadre, d'une part que l'algorithme d'Audsley est optimal, et d'autre part, que l'algorithme développé par Vestal minimise la vitesse du processeur nécessaire pour ordonnancer fiablement les tâches . Nous avons également adapté la méthode d'analyse de sensibilité développée par Bini pour déterminer les variations admissibles des durées d'exécution des tâches, ainsi qu'introduit la notion de gigue sur activation, ce qui n'avait pas encore été fait à notre connaissance.

Enfin, dans le chapitre 5, nous nous sommes intéressés aux systèmes multiprocesseurs ordonnancés par des algorithmes d'ordonnancement semi-partitionnés. Cette classe d'algorithme utilise une stratégie par partitionnement pour répartir les tâches parmi les processeurs tout en autorisant la migration des tâches qui ne peuvent pas être affectées à un processeur donné sans violation d'échéance. Nous avons développé des algorithmes semi-partitionnés en priorité dynamique (EDF), autorisant la migration des tâches, mais interdisant la migration des instances afin de limiter le surcoût processeur induit par les migrations. Le principe de nos algorithmes est de dupliquer chaque tâche migrante sur chacun des processeurs devant exécuter au moins une instance de cette tâche, en utilisant une tâche multiframe. Nous avons comparé nos algorithmes avec l'algorithme First Fit Decreasing (systèmes partitionnés) et avec un autre algorithme de semi-partitionnement, développé par Shinpei Kato (qui autorise les migrations des instances de tâches en cours d'exécution). Nous avons constaté que notre algorithme fournissait de bon résultats par rapport à l'algorithme First Fit Decreasing, et des performances proches de celle de l'algorithme de Shinpei Kato, qui est un algorithme qui autorise la migration des instances, donc intrinsèquement plus puissant. Nous avons également étudier l'influence que pouvait avoir certains paramètres, tels que le nombre de processeurs ou le nombre de frames des tâches multiframes.

Suite à ces travaux, les perspectives envisagées sont d'étendre l'algorithme de minimisation du nombre de processeurs par :

- la prise en compte de nouvelles contraintes, comme la pré-affectation de tâches, ou la prise en compte de contraintes supplémentaires ;
- l'intégration de nos travaux concernant le modèle de tâches multicritiques ;
- l'intégration de nos travaux sur les algorithmes semi-partitionnés, à travers l'adaptation de la méthode de validation pour pouvoir tenir compte de tâches multiframes d'une part, et par l'utilisation d'une politique d'ordonnancement utilisant des priorités fixes plutôt que dynamiques d'autre part.

A plus long terme, l'intégration de tous ces travaux dans l'algorithme de minimisation du nombre de processeurs fournirait alors un algorithme puissant d'aide à la conception et à la validation de systèmes temps réel sur des architectures distribuées. Il serait alors intéressant de l'implémenter et de proposer cette implémentation en tant qu'outil d'aide à la conception et à la validation à destination des concepteurs de systèmes temps réel.

Annexes

Notations

Voici un récapitulatif des notations utilisées dans le cadre de cette thèse :

- $A[1 \dots n, 1 \dots m]$: tableau de répartition des instances des tâches sur les processeurs
- $A[i, j]$: nombre d'instances de la tâche τ_i sur le processeur P_j
- B_i : facteur de blocage de τ_i (durée maximum d'attente avant l'obtention d'une ressource)
- C_i : temps d'exécution de la tâche τ_i
- $C_i(\ell)$: temps d'exécution de la tâche τ_i au niveau de criticité ℓ
- $\mathbb{C}_i$: vecteur des i tâches les plus prioritaires
- D_i : échéance relative de la tâche τ_i
- $d_{i,j}$: échéance absolue de la $j^{\text{ème}}$ instance de la tâche τ_i
- $dbf(\tau_i, t)$: *demand bound function* de τ_i sur l'intervalle $[O, t[$
- $\widehat{dbf}(\tau_i, t)$: *demand bound function* pour une tâche migrante dans le cadre du scénario compact
- $\widehat{\widehat{dbf}}(\tau_i, t)$: *demand bound funciton* pour une tâche migrante dans le cadre du scénario avec prise en compte du schéma de répartition
- $\Delta(\tau_i, \tau)$: facteur d'échelle critique de τ_i lorsque l'ensemble des tâches plus prioritaire est τ
- Δ^* : facteur d'échelle critique du système
- Δ_a : facteur d'échelle critique du résultat obtenu par l'algorithme d'Audsley
- Δ_v : facteur d'échelle critique du résultat obtenu par l'algorithme de Vestal
- Δ_i : facteur d'échelle critique de la tâche τ_i
- $\Delta_{i,j}$ facteur d'échelle critique de la tâche τ_i au niveau de priorité j
- $\delta C_k^{\max}$: modification du temps d'exécution qui peut être appliquée à la tâche τ_k sans modifier l'ordonnançabilité du système
- $I_i(t)$: interférence que subit la tâche τ_i durant un interval de durée t
- J_i : gigue sur activation de la tâche τ_i
- K : nombre de frames des tâches multiframes
- $LB(J_i)$: borne inférieure de la gigue de τ_i
- $LB(Tr_i)$: borne inférieure du temps de réponse de τ_i

- L_i : niveau de criticité de la tâche τ_i
- λ : facteur d'échelle
- λ_ℓ : facteur d'échelle pour le niveau de criticité ℓ
- Pl_i : pool numéro i
- P_i : processeur numéro i
- π_i : priorité de la tâche τ_i, avec $\pi_i = 0$ désignant la plus forte priorité
- $rbf(\tau_i, t)$: *request bound function* de τ_i sur l'intervalle $[O, t[$
- $S(\tau_i, t)$: nombre d'unité processeur utilisé par τ_i dans l'intervalle $[0, t[$
- S_i : ensemble de points d'ordonnancement de la tâche τ_i
- $Sched(P_i)$: ensemble des points d'ordonnancement pour le processeur P_i
- $\Sigma^{-1}(i)$: ensemble des prédécesseurs de τ_i
- Σ_i^{*-NP} : ensemble des messages émis par les prédécesseurs de τ_i qui ne sont pas placés mais qui n'appartiennent pas au pool courant
- Σ_i^{*-C} : ensemble des prédécesseurs de τ_i situés sur le même processeur que τ_i
- Σ_i^{-NP} : ensemble des messages émis par les prédécesseurs de τ_i qui ne sont pas placés mais qui appartiennent au pool courant
- Σ_i^{-P} : ensemble des messages émis par les prédécesseurs de τ_i, qui sont placés sur un processeur différent au moment de l'analyse de τ_i
- Θ_i : tâches restant à placer sur le pool Pl_i
- T_i : période de la tâche τ_i
- Tr_i : temps de réponse de la tâche τ_i
- $Tr_{i,j}$: temps de réponse de la $j^{\text{ème}}$ instance de τ_i
- τ_i : tâche numéro i
- u_i : facteur d'utilisation de la tâche τ_i
- U : facteur d'utilisation du système de tâche
- $W(t)$: travail processeur sur l'intervalle $[0, t[$
- $W_i(t)$: travail processeur des tâches de priorité supérieure ou égale à π sur l'intervalle $[0, t[$

Accronymes

Voici un récapitulatif des différents accronymes utilisés dans le cadre de cette thèse :
- AMNP : Algorithme de Minimisation du Nombre de Processeurs
- APOC : Algorithme de Placement et d'Ordonnancement Conjoints
- dbf : Demand Bound Function
- DP : Dynamic Priority
- EDF : Earliest Deadline First
- EDZL : Earliest Deadline Zero Laxity
- FJP : Fixed Job Priority
- FTP : Fixed Task Priority
- g-EDF : global EDF
- LL : Last Laxity
- PMS : Project Management and Scheduling
- PPH : Protocole à Priorité Héritée
- PPP : Protocole à Priorité Plafond
- rbf : Request Bound Function
- ROADEF : Recherche Opérationnelle et d'Aide à la Décision
- RTNS : Real-Time and Network Systems
- RTS : Real-Time Systems

Index

Bibliographie

[AB08] B. Andersson and K. Bletsas. Sporadic multiprocessor scheduling with few preemption. In *2008 Euromicro Conference on Real-Time Systems*, pages 243–252, July 2008.

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

[ABR+93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8 :284–292, 1993.

[ARI97] ARINC. Avionics application software standard interface. *ARINC Spec*, 653, 1997.

[Ari09] Arinc. *Aircraft Data Network, Part 7, Avionics Full-Duplex Switched Ethernet Network*. Arinc, 2009.

[AS00] J. H. Anderson and A. Srinivasan. Early-release fair scheduling. In *12th Euromicro Conference on Real-Time Systems*, pages 35–43, 2000.

[AT06] B. Andersson and E. Tovar. Multiprocessor scheduling with few preeptions. In *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334, 2006.

[Aud91] N.C. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start time. Technical Report Technical Report YCS 164, Dept. Computer Science, University of York, UK, December 1991.

[AUT] AUTOSAR. Offical website. http ://www.autosar.org.

[Bak05a] T. P. Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and edf scheduling for hard real time. Technical Report TR-050601, Department of Computer Science, Florida State University, 2005.

[Bak05b] T.P. Baker. An analysis of edf schedulability on a multiprocessor. *IEEE Transactions on Parallel and Distributed Systems*, 16(8) :760–768, August 2005.

[Bar97] M. Barabanov. *A Linux-based Real-Time Operating System*. PhD thesis, New Mexico Institute of Mining and Technology, 1997.

[BAW95] A. Burns, N. Audsley, and A. Wellings. Real-time distributed computing. In *Proceeding of the 5th IEEE Computer Society Workshop – Future Trends of Distributed Computing Systems*, volume 0, pages 34–40. IEEE Computer Society, 1995.

[BB04a] 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, Washington, DC, USA, 2004. IEEE Computer Society.

[BB04b] E. Bini and G.C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11) :1462–1473, Nov. 2004.

[BB06] T. P. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of real-time and embedded systems*. Chapman Hall/CRC Press, 2006.

[BB08] S. Baruah and T. P. Baker. Schedulability analysis of global edf. *Real-Time Systems*, 38 :223–235, 2008.

[BB09] E. Bini and G. C. Buttazzo. The space of edf deadlines : the exact region and a convex approximation. *Real-Time Systems*, 41 :27–51, 2009.

[BCB08] T. P. Baker, M. Cirinei, and M. Bertogna. EDZL scheduling analysis. *Real-Time Systems*, 40(3) :264–289, December 2008.

[BCGM99] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Time-Critical Computing Systems*, 17(1) :5–22, 1999.

[BCPV96] S. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress : a notion of fairness in resource allocation. *Algorithmica*, 15(6) :600–625, June 1996.

[BDNB08] E. Bini, M. Di Natale, and G. Buttazzo. Sensitivity analysis for fixed-priority real-time systems. *Real-Time Systems*, 39(1–3) :5–30, 2008.

[BDV04] A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the ada ravenscar profile in high integrity systems. *Ada Letters*, 24 :1–74, 2004.

[BFR75] P. Bratley, M. Florian, and P. Robillard. Scheduling with earliest start and due date constraints on multiple machines. *Naval Research Logistic Quaterly*, 22 :165–173, 1975.

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

[Bla09] J-P. Blanquart. Sûreté de fonctionnement des systèmes embarqués critiques : les enjeux industriels (domaine spatial). In *Ecole d'Eté Temps réel*, 2009.

[BLDB06] R. Bril, J. Lukkien, R. Davis, and A. Burns. Message response time analysis for ideal controller area network (can) refuted. In *Proceedings of the 5th International Workshop on Real-Time Networks*, 2006.

[BLOS95] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers, IEEE Transactions on*, 44(12) :1429–1442, December 1995.

[Bou07] L. Bougueroua. *Conception de systems temps réel déterministes en environnement incertain*. PhD thesis, Université Parix XII, March 2007.

[BRC09] P. Balbastre, I. Ripoll, and A. Crespo. Period sensitivity analysis and D-P domain feasibility region in dynamic priority systems. *Journal of Systems and Software*, 82(7) :1098–1111, 2009.

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

[But97] Giorgio C. Buttazzo. *Hard real-time computing systems : Predictable scheduling algorithms and applications*. Kluwer Academic Publishers, 1997.

[BV08] S. Baruah and S. Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, pages 147–155, Washington, DC, USA, 2008. IEEE Computer Society.

[CLAL02] S. Cho, S-K. Lee, S. Ahn, and K-J. Lin. Efficient real-time scheduling algorithms for multiprocessor systems. *IEICE Transactions on Communications*, 85 :807–813, 2002.

[DBBL07] R. Davis, A. Burns, R. Bril, and J. Lukkien. Controller area network (can) schedulability analysis : Refuted, revisited and revised. *Real-Time Systems*, 35(3) :239–272, April 2007.

[Der74] Michael L. Dertouzos. Control robotics : The procedural control of physical processes. In *IFIP Congress*, pages 807–813, 1974.

[DGRR10] F. Dorin, J. Goossens, P. Richard, and M. Richard. Schedulability analysis of multiple criticality real-time tasks. In *Proceedings of the 12th International Workshop on Project Management and Scheduling*, pages 167–170, April 2010.

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

[DMYGR10] François Dorin, Patrick Meumeu Yomsi, Joël Goossens, and Pascal Richard. Semi-partitioned hard real-time scheduling with restricted migrations upon identical

multiprocessor plarforms. In *Real Time and Networks Systems*, 2010. prépublication.

[DRGR08] F. Dorin, M. Richard, E. Grolleau, and P. Richard. Minimizing the number of processors for real-time distributed systems. In *Proceedings of the 16th International Conference on Real-Time and Network System*, pages 121–130, 2008.

[DRGR09] F. Dorin, M. Richard, E. Grolleau, and P. Richard. Minimisation du nombre de processeurs pour les systèmes temps réel distribués. In *10ème conférence de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF'09)*, pages 308–309, February 2009.

[DRRG09] F. Dorin, P. Richard, M. Richard, and J. Goossens. Uniprocessor schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities. In *Proceedings of the 17h International Conference on Real-Time and Network Systems*, pages 13–22, 2009. Best Student Paper Award.

[ER08] F. Eisenbrand and T. Rothvoß. Static-priority Real-time Scheduling : Response Time Computation is NP-hard. In *Proceedings of the 2008 Real-Time Systems Symposium*, pages 397–406, 2008.

[ER10] F. Eisenbrand and T. Rothvoß. EDF-schedulability of synchronous periodic task systems is coNP-hard. In *Proceedings of the 2010 ACM-SIAM Symposium on Discrete Algorithms*, 2010.

[FBB06] N. Fisher, S. Baruah, and T. P. Baker. The partitioned scheduling of sporadic tasks according to static-priorities. *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 118–127, 2006.

[FMB+09] S. Fürst, J. Mössinger, S. Bunzel, T. Weber, F. Kirschke-Biller, P. Heitkämper, G. Kinkelin, K. Nishikawa, and K. Lange. *AUTOSAR - A wordwide standard is on the road*, 2009.

[GBF02] J. Goossens, S. Baruah, and S. Funk. Real-time scheduling on multiprocessors. In *In Proceedings of the 10th Internation Conference on Real-Time Systems*, pages 189–204, 2002.

[GFB03] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2–3) :187–205, 14 2003.

[GGH98] Palencia Gutiérrez, Gutiérrez García, and Gonzalez Harbour. Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems. In *Proceedings of the 10th Euromicro Wirkshop on Real Time Systems*, pages 35–44, June 1998.

[HL92] K. S. Hong and J. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computuers*, 41(10) :1326–1331, 1992.

[Hla04] Pierre-Emmanuel Hladik. *Ordonnançabilité et placement des systèmes temps réel distribués, prémptifs et à priorité fixes.* PhD thesis, Université de Nantes, 2004.

[Hor74] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1) :177–185, 1974.

[IEE04] IEEE. *IEEE Std 1003.1,2004 Edition*, 2004 edition, 2004. Posix Specification.

[ISO94a] ISO. *Road vehicles – Low-speed serial data communication – Part 2 : Low-speed controller area network (CAN).* ISO International Standard 11519-2, 1994.

[ISO94b] ISO. *Road vehicles – Low-speed serial data communication – Part 3 : Vehicle area network (VAN).* ISO International Standard 11519-3, 1994.

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

[Kai82] C. Kaiser. Exclusion mutuelle et ordonnancement par priorité. *Technique et Science Informatiques*, 1 :59–68, 1982.

[KAS93] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Transactions on Software Engineering*, 19(9) :920–934, 1993.

[KRP+93] M. Klein, T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour. *A Practitioner's Handbook for Real-Time Analysis : Guide to Rate Monotonic Analysis for Real-Time Systems.* Kluwer Academic Publishers, 1993.

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

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

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

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

[LBJ+95] S-S. Lim, Y. H. Bae, G. T. Jang, B-D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, S-M. Moon, and C. S. Kim. An accurate worst case timing analysis for risc processors. *IEEE Transactions on Software Engineering*, 21(7) :593–604, July 1995.

[Leh90] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Proceedings of the 11th Real-Time Systems Symposium*, pages 201–209, December 1990.

[LGDG00] J. M. López, M. García, J. L. Díaz, and D. F. García. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, 2000.

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

[LM80] J. Leung and M. Merill. A note on preemptive scheduling of periodic, real-time tasks. *Information processing letters*, 11(3) :115–118, 1980.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm-exact characterization and average case behavior. In *Proceedings of the Real-Time System Sumposium*, pages 166–171, December 1989.

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

[MA98] Y. Manabe and S. Aoyagi. A feasibility decision algorithm for rate monotonic and deadline monotonic scheduling. *Real-Time Systems*, 14(2) :171–181, Marsh 1998.

[Mok83] A. K. Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

[NSKT98] K. Nabeshimak, T. Suzudo, Suzuki K., and E. Türcan. Real-time nuclear power plant monitoring with neural network. *Journal of Nuclear Science and Technology*, 35(2) :93–100, 1998.

[Ose05] Osek. *Osek Operating System*. Osek, 2005.

[Owe97] P. Owezarski. Modélisation, conception et implémentation d'applications multimédias temporellement contraintes. In *Ecole d'Eté Temps Réel*, 1997.

[PDB97] S. Punnekkat, R. Davis, and A. Burns. *Advances in Computing Science*, chapter Sensitivity analysis of real-time task sets, pages 72–82. Springer-Verlag, 1997.

[PHK+05] M. Park, S. Han, H. Kim, S. Cho, and Y. Cho. Comparison of deadline-based scheduling algorithms for periodic real-time tasls on multiprocessor. *IEICE - Transactions on Information and Systems*, 88(3) :658–661, 2005.

[RCM96] I. Ripoll, A. Crespo, and A. Mok. Improvement in feasibility testing for real-time tasks. *Real-Time Systems*, 11(1) :19–39, July 1996.

[RGR08] A. Rahni, E. Grolleau, and M. Richard. Feasibility analysis of non-concrete real-time transactions with edf. In *Proceedings of the 16th International Conference on Real-Time and Network Systems*, pages 109–117, October 2008.

[RHE06] R. Racu, A. Hamann, and R. Ernst. A formal approach to multi-dimensional sensitivity analysis of embedded real-time systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 3–12, 2006.

[RHE08] R. Racu, A. Hamann, and R. Ernst. Sensitivity analysis of complex embedded real-time systems. *Real-Time Systems*, 39(1) :31–72, August 2008.

[Ric02] Michaël Richard. *Contribution à la validation des systèmes temps réel distribués : ordonnancement à priorités fixes & placement.* PhD thesis, Université de Poitiers, 2002.

[Riv] Wind River. Vxworks, publisher's website. http ://www.windriver.com.

[Ros] J-P. Rosen. Ada 95 pour le temps réel et les systèmes distribués.

[SL96] Jun Sun and Jane Liu. Bounding the end-to-end response times of tasks in a distributed real-time system using the direct synchronization protocol. Technical Report UIUCDCS-R-96-1949, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1996.

[SLPH+05] J. Souyris, E. Le Pavec, G. Himbert, V. Jégu, G. Borios, and R. Heckmann. computing the wcet of an avionics program by abstract interpretation. In *WCET*, pages 15–18, 2005.

[Spu96] M. Spuri. Analysis of deadline scheduled real-time systems. Technical Report 2272, INRIA Research Report, 1996.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols : An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9) :1175–1185, 1990.

[SS94] M. Spuri and J. A. Stankovic. How to integrate precedence constraints and shared resources in real-time scheduling. *IEEE Transactions on Computers*, 43(12) :1407–1412, December 1994.

[TB94] K. Tindell and A. Burns. Guaranteeing message latencies on Controller Area Network (CAN). In *Proceedings of 1st international CAN Conference*, September 1994.

[Tin94] K. W. Tindell. *Fixed priority scheduling of hard real-time systems.* PhD thesis, University of York, 1994.

[TSH+03] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software systems. In *Proceedings of the Performance and Dependability Symposium*, 2003.

[Ves94] S. Vestal. Fixed-priority sensitivity analysis for linear compute time models. *IEEE Transactions on Software Engineering*, 20(4) :308–317, 1994.

[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, Washington, DC, USA, 2007. IEEE Computer Society.

[Wal97] J Walter. Le temps réel en sidérurgie à sollac florange. In *Ecole d'Eté Temps Réel*, 1997.

[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) :1–53, 2008.

[ZB09] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *IEEE Transactions on Cumputers*, 58(9) :1250–1258, September 2009.

Résumé

Dans nos travaux, nous nous sommes intéressés aux politiques d'ordonnancement en-ligne, pour lesquelles l'ordre d'exécution des tâches est déterminé sur la base des tâches actives dans le système et sans connaissance a priori des tâches qui arriveront dans le futur. Cette problématique s'étend aux réseaux informatiques pour ordonnancer les messages sur le réseau informatique interconnectant les différents calculateurs du système temps réel.
La première contribution concerne le placement et ordonnancement simultanés des tâches dans un système temps réel distribué, permettant de minimiser le nombre de processeurs nécessaires pour respecter les spécifications temporelles des tâches. La seconde contribution concerne les tâches à criticité multiple. L'objectif de ce modèle est de prendre en compte la notion de criticité des différentes tâches, comme par exemple dans la norme DO-178B utilisée en aéronautique.
Enfin, nous nous sommes intéressés aux systèmes multiprocesseurs ordonnancés par des algorithmes d'ordonnancement semi-partitionné. Cette classe d'algorithme utilise une stratégie par partitionnement pour répartir les tâches parmi les processeurs tout en autorisant la migration des tâches qui ne peuvent pas être affectées à un processeur donné sans violation d'échéance.

Mots-clés : systèmes temps réel, ordonnancement, analyse d'ordonnançabilité, systèmes monoprocesseurs, multiprocesseurs et distribués, tâches à criticité multiple, algorithmes semi-partitionnés,

---

Abstract

In our works, we were interested by on-line scheduling algorithms, for which the order of the execution of the tasks is determined only by the knowledge of the actvie task set, that is to say without any knowledge of the tasks which may arrive in the future. This issue extends on networks to schedule messages on the network which interconnects the different calculators of the real-time system.
The first contribution deals with the task allocation on the processors and the priority assignment of the tasks in a distributed real-time system, which minimizes the number of processors needed to meet the task temporal specification. The second contribution is about multiple criticality tasks. The aim of this model is to address the notion of criticity, as defined in the DO-178B aeronautical standard.
Finally, we were interested by multiprocessor systems scheduled using a semi-partitioned algorithm. This kind of algorithm uses a partitioned strategy in order to share the tasks among the processors and allow migrations for tasks which cannot be assigned to a given processor without missing a deadline.

Keywords : real-time systems, scheduling, schedulability analysis, uniprocessor, multiprocessor and distributed systems, multiple criticality tasks, semi-partitioned algorithms

---

Secteur de recherche : Informatique

LABORATOIRE D'INFORMATIQUE SCIENTIFIQUE ET INDUSTRIELLE
Ecole Nationale Supérieure de Mécanique et d'Aérotechnique
Téléport 2 – 1, avenue Clément Ader – BP 40109 – 86961 Chasseneuil-Futuroscope Cédex
Tél : 05.49.49.80.63 – Fax : 05.49.49.80.64