Analyse et ordonnancement d'un système hiérarchique virtualisé composé d'applications temps réel strictes

Tristan Fautrel

Université Gustave Eiffel

Analyse et ordonnancement d'un système hiérarchique virtualisé composé d'applications temps réel strictes

Thèse de doctorat de l'Université Gustave Eiffel

École doctorale MSTIC n°532 (Mathématiques et STIC)
Spécialité de doctorat : Informatique
Unité de recherche : UMR CNRS 8049 LIGM

Thèse présentée et soutenue à l'Université Gustave Eiffel, le 06/05/2021, par

Tristan FAUTREL

Composition du Jury

Joël GOOSSENS Professeur, ULB	Président
Marilyne CHETTO Professeur, Université de Nantes	Rapporteur
Laurent PAUTET Professeur, Télécom Paristech	Rapporteur
Liliana CUCU-GROSJEAN Chargée de recherche (HDR), INRIA	Examinatrice
Laurent GEORGE Professeur, ESIEE Paris	Directeur de thèse

Encadrement de la thèse

Frédéric FAUBERTEAU Enseignant-chercheur, ESILV	Co-Encadrant
Thierry GRANDPIERRE Professeur associé, ESIEE Paris	Co-Encadrant

UNIVERSITÉ PARIS-EST

THÈSE DE DOCTORAT

Analyse et ordonnancement d'un système hiérarchique virtualisé composé d'applications temps réel strictes

---

Tristan Fautrel

Rapporteurs	Maryline Chetto	Professeur	Université de Nantes
	Laurent Pautet	Professeur	Télécom Paristech
Examinateurs	Liliana Cucu-Grosjean	Chargée de recherche (HDR)	INRIA
	Joël Goossens	Professeur	ULB
Directeur	Laurent George	Professeur	ESIEE Paris
Co-encadrants	Frédéric Fauberteau	Enseignant-chercheur	ESILV
	Thierry Grandpierre	Professeur associé	ESIEE-Paris

Cette thèse d'informatique a été réalisée dans les laboratoires

de

LIGM
École Doctorale MSTIC

6 mai 2021

« *I've told you once I've told you a thousand times*
No regrets and no remorse
No four A.M. whiskey soaked wisdom or bloody
knuckled politics
Do I regret and not a single moment will I ever repent

You may say I have lost to a better man
This may be true I cannot protest or lie
Yet maybe one who did not dare to be wrong or
even to be right

To those who did not dare to sing out of tune
Or sing a different song
To march to the beat of a different drum and speak
the truths others fear
Just give me one thing to live or die for

So here's to comrades near and far
So raise a glass, raise your voices
Years have passed some would say
They have not been kind
Yet these are the scars of war
And we remain yet we stand
Bloodied yet unbowed

What is the standard with which I bear
What flag do we fly when marching to war
Only a nation that dare not speak its name
Nor can ever shed its pain

So here's to comrades near and far
Raise a glass, raise hell
Years have passed closer to the grave
But this is the song we chose to sing
To the bitter end, to the end »

Primordial - Bloodied yet unbowed

Résumé

Analyse et ordonnancement d'un système hiérarchique virtualisé composé d'applications temps réel strictes

Dans le contexte des villes intelligentes, le concept de véhicule autonome prend de l'importance dans l'objectif de mener à bien des missions de surveillance pour la protection des infrastructures. Dans ce contexte, les véhicules aériens sans pilote (drones) embarquent plusieurs applications (contrôle commande, communication et vidéo) de criticités différentes avec des contraintes de certification associées souvent hétérogènes. L'application de contrôle-commande ayant le niveau de criticité le plus élevé doit être certifiée au plus haut niveau d'exigence. Lorsque plusieurs applications de criticité différente s'exécutent sur la même plate-forme matérielle, l'application la plus critique impose, par contagion aux applications les moins critiques, une certification du même niveau d'exigence que celui de l'application la plus critique. Ce qui n'est pas toujours possible, compte tenu de la complexité des applications, et peut conduire à un coût de certification qui peut être important. Une solution à ce problème consiste à utiliser un système de virtualisation basé sur un hyperviseur certifié, assurant une isolation spatiale et temporelle des applications. Chaque application s'exécute alors dans une machine virtuelle (VM) dédiée. L'activation des VMs est gérée par l'hyperviseur en charge de cette isolation.

D'un point de vue de l'ordonnancement, cela forme un système hiérarchique à deux niveaux. Les VMs sont ordonnancées par l'hyperviseur au premier niveau, les tâches de chaque VM sont ordonnancées au second niveau par un algorithme d'ordonnancement spécifique. Nous considérons dans cette thèse un ordonnancement à priorité fixe au second niveau.

Nos travaux s'intègrent dans le contexte du projet de drone de surveillance CEOS. Dans ce projet, le but est de créer un drone autonome de surveillance à but non militaire. Il s'agit d'un projet FUI avec différents partenaires tels que l'INRIA, Enedis, EDF, Sysgo, Alerion, Thalès, ou encore l'aéroport de Caen. Ce drone est défini pour trois cas d'usage : (i) la surveillance des lignes électriques dans le réseau ENEDIS, (ii) la surveillance de conduites d'eau dans les montagnes pour EDF et (iii) la surveillance de clôtures dans l'aéroport de Caen. Pour le projet CEOS, un hyperviseur de type 1 développé par la société Sysgo (PikeOs) a été choisi. PikeOs est certifié DAL-A pour divers standards avioniques tels que ARINC 653 et MILS. C'est grâce à ses certifications que cet hyperviseur a été choisi. Une de ses particularités est qu'il est possible de maîtriser l'ordonnancement de niveau 1 des VMs. Pour le niveau 1, l'ordonnancement des machines virtuelles à l'aide d'une table d'ordonnancement doit être déterminé au préalable, aucun algorithme n'étant imposé. Concernant les tâches exécutées par les VMs, leur ordonnancement est déterminé par le système d'exploitation temps réel associé à la machine virtuelle. Il est également possible d'introduire des tâches directement dans PikeOS qui seront alors ordonnancées par un ordonnancement à priorité fixe.

La liberté du choix d'algorithme pour l'ordonnancement de premier niveau implique plusieurs questions : «comment ordonnancer les machines virtuelles pour permettre aux tâches de respecter leurs échéances ?». Dans les conditions d'un ordonnancement déterminé à l'avance pour les machines virtuelles, «quel ordonnancement donnerait un bon taux de succès pour les tâches ?». C'est à ces différentes questions que nous répondons tout au long de cette thèse, en commençant par formaliser et analyser le modèle d'ordonnancement hiérarchique à deux niveaux. Grâce à cette analyse, nous déterminons un premier ordonnancement pour les machines

virtuelles garantissant un taux d'utilisation pour la VM selon une approche à périodicité stricte, pour des périodes d'activation des VMs harmoniques. Cette approche conduit à activer les VMs selon un motif d'activation constitué d'un seul slot, activé périodiquement.

Ce seul algorithme ne permet pas toujours de répondre à la question car toutes les machines virtuelles ne sont pas ordonnançables grâce à ce seul algorithme, c'est pourquoi nous proposons un second algorithme, appelé algorithme proportionnel itératif, qui s'inspire de l'ordonnancement P-fair. Il nous permet ainsi de compléter la table d'ordonnancement des machines virtuelles qui ne sont pas ordonnançables par la première approche.

Concernant l'ordonnancement de niveau 2, nous considérons un ordonnancement à priorité fixe au niveau des tâches. Les tâches sont modélisées par des tâches sporadiques à échéances arbitraires. Nous établissons les conditions d'ordonnançabilité des tâches pour un motif arbitraire d'activation périodique de la VM en charge de l'exécution de ces tâches. Nous nous intéressons ensuite à l'ordonnancement de niveau 2 Dual Priority pour obtenir un meilleur taux de succès pour des tâches ordonnancées dans un modèle hiérarchique à deux niveaux. Cet ordonnancement bien que non optimal a un très bon taux de succès même pour des charges élevées. Cet ordonnancement nécessite d'affecter deux priorités fixes à la tâche : la première à la date de son activation, la seconde à une de ses échéances de promotion. Nous proposons deux algorithmes permettant de choisir de manière efficace les deux priorités fixes et l'échéance de promotion.

Pour obtenir les différents taux de succès des algorithmes proposés, nous avons développé un simulateur d'ordonnancement temps réel pendant la thèse. L'outil développé se veut générique et extensible. De plus, pour valider les algorithmes présentés dans cette thèse, nous avons également développé un outil pour le projet CEOS qui permet de déterminer la table d'ordonnancement des machines virtuelles à utiliser pour PikeOS. Les recherches ainsi présentées ne sont pas uniquement théoriques mais visent à être appliquées dans des projets concrets utilisant PikeOS pour l'administration de machines virtuelles.

Abstract

Analysis and scheduling of hierarchical virtualized systems composed of hard real-time applications

In the context of smart cities, the concept of autonomous vehicle is gaining importance in the objective of carrying out surveillance missions for the protection of infrastructures. In this context, unmanned aerial vehicles (drones) embed several applications (control command, communication and video) of different criticality with often heterogeneous associated certification constraints. The instrumentation and control application with the highest level of criticality must be certified at the highest level of requirement. When multiple applications of different criticality run on the same hardware platform, the most critical application contagiously imposes certification on the less critical applications to the same level of requirement as that of the most critical application. This is not always possible given the complexity of the applications and can lead to a certification cost which can be significant. One solution to this problem is to use a virtualization system based on a certified hypervisor, providing spatial and temporal isolation of applications. Each application then runs in a dedicated virtual machine (VM). The activation of VMs is managed by the hypervisor in charge of this isolation.

From a scheduling point of view, this forms a two-level hierarchical system. VMs are scheduled by the hypervisor at the first level, the tasks of each VM are scheduled at the second level by a specific scheduling algorithm. We consider in this thesis a fixed priority scheduling at the second level.

Our work fits into the context of the CEOS surveillance drone project. In this project, the goal is to create an autonomous surveillance drone for non-military purposes. This is a FUI project with various partners such as INRIA, Enedis, EDF, Sysgo, Alerion, Thalès, Caen airport. This drone is defined for three use cases: (i) monitoring power lines in the ENEDIS network, (ii) monitoring water pipes in the mountains for EDF and (iii) monitoring fences in the Caen airport.

For the CEOS project, a type 1 hypervisor developed by the Sysgo company (PikeOs) was chosen. PikeOs is certified DAL-A for various avionics standards such as ARINC 653 and MILS. It is thanks to its certifications that this hypervisor was chosen. One of its peculiarities is that it is possible to master the level 1 scheduling of VMs. For level 1, the scheduling of virtual machines using a scheduling table must be determined beforehand, no algorithm is imposed. Regarding the tasks executed by the VMs, their scheduling is determined by the real-time operating system associated with the virtual machine. It is also possible to introduce tasks directly into PikeOS which will then be scheduled by a fixed priority scheduling.

The freedom to choose an algorithm for first-level scheduling involves several questions: How to schedule VMs so that all tasks in their VM meet their deadlines? Under the conditions of predetermined scheduling for virtual machines, which scheduling would give a good success ratio for the scheduling of tasks?

It is to these different questions that we answer throughout this thesis, starting by formalizing and analyzing the two-level hierarchical ordering model. Thanks to this analysis, we determine a first scheduling for the virtual machines for the VM according to a strict periodicity approach guaranteeing a utilization rate, for VMs with harmonic periods of activation. This approach leads to activating the VMs according to an activation pattern consisting of a single slot, activated periodically.

This single algorithm does not always answer the question because not all virtual machines are schedulable thanks to this single algorithm. That is why we propose

a second algorithm, called Proportional Iterative Algorithm, which is inspired by the P-Fair scheduling. It thus allows us to complete the scheduling table of virtual machines which are not schedulable by the first approach.

Regarding the scheduling of tasks at level 2, we consider a fixed priority scheduling. Tasks are modeled as sporadic tasks with arbitrary deadlines. We establish the conditions for the scheduling of tasks for an arbitrary VM pattern of activation in charge of running these tasks. We then turn to Dual Priority level 2 scheduling to achieve a better success ratio for tasks scheduled in a two-level hierarchical model. This scheduling, although not optimal, has a very good success ratio even for high loads. This scheduling requires assigning two fixed priorities to the task, the first at the activating the task, the second at a promotion deadline. We propose two algorithms to efficiently choose the two fixed priorities and the promotion deadline.

To obtain the different success ratio of the proposed algorithms, we developed a real-time scheduling simulator during the thesis. The tool developed is intended to be generic and extensible. In addition, to validate the algorithms presented in this thesis, we have also developed a tool for the CEOS project which makes it possible to determine the virtual machine scheduling table to be used for pikeOS. The research presented in this way is not only theoretical but aims to be applied in real projects for any PikeOS user who uses virtual machines.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Dans ce premier chapitre, nous commençons par présenter la problématique de cette thèse ainsi que son contexte, le projet CEOS, qui nous a poussés à développer le modèle utilisé dans la thèse. Nous décrivons ensuite les différentes étapes de cette thèse pour répondre à la problématique.

Sommaire

1.1 Problématique

Avec l'évolution de la puissance des processeurs, il est de plus en plus fréquent de n'utiliser qu'un seul processeur et ses différents cœurs pour différentes applications. Un problème majeur se pose alors : «comment répartir les différentes ressources du processeur entre les différentes applications tout en s'assurant que toutes les tâches de chaque application se terminent avant un délai fixé ?».

Pour introduire précisément le contexte, nous fixons quelques éléments de terminologie. Ici, les applications sont un regroupement de tâches, qui correspondent à des programmes. Ainsi, chaque programme, donc tâche, doit terminer son exécution avant un délai donné, appelé échéance. Si l'on souhaite isoler ces tâches du reste des applications, il existe deux approches majeures. La première consiste en l'installation des applications dans des conteneurs distincts sur un même système d'exploitation. La seconde consiste en la séparation des applications dans différentes machines virtuelles exécutant chacune leur propre système d'exploitation. Cette dernière approche implique l'utilisation d'un hyperviseur entre la couche matérielle et les machines virtuelles pour gérer l'ordonnancement de celles-ci.

La première solution est moins gourmande en ressources, du fait de l'utilisation d'un seul système d'exploitation, mais cela impose à toutes les applications d'utiliser le même système, et surtout d'avoir un système d'exploitation capable de garantir l'isolation spatiale et temporelle entre les différentes applications. Dans le cadre de cette thèse, nous choisissons la deuxième solution, à savoir l'utilisation des machines virtuelles. Sur des projets industriels, cela permet de n'assurer l'isolation spatiale et temporelle qu'à travers l'hyperviseur, et cela permet une plus grande souplesse pour le développement des applications. En effet, chaque équipe qui travaille sur une application ne doit s'occuper que de sa propre application et peut travailler à son rythme, sans avoir à interagir avec les autres équipes, sur le système d'exploitation de son choix qui ne supporte pas forcément la containérisation (*e.g.* FreeRTOS). Le seul moment où les équipes devront s'organiser sera au moment de l'intégration des différentes machines virtuelles sur l'hyperviseur. Un tel comportement, notamment pouvoir travailler sur des systèmes d'exploitation différents, n'est pas possible à l'aide de conteneurs.

Pour garantir l'isolation spatiale et temporelle entre les différentes machines virtuelles, il existe de nombreuses certifications telles que DO-178C, DAL-A et ARINC 653. Ces certifications permettent de garantir qu'une défaillance d'une machine virtuelle n'impacte pas les autres. Il faut également pouvoir garantir que les machines virtuelles ne peuvent pas influer involontairement l'une sur l'autre. Par exemple, il ne faut pas qu'une machine virtuelle puisse écrire dans l'espace mémoire d'une autre machine virtuelle sans que l'espace mémoire en question ne soit volontairement partagé. Tous ces éléments font partie de l'isolation.

Il est également intéressant de pouvoir certifier chaque machine virtuelle indépendamment l'une de l'autre afin de ne pas avoir à certifier les applications au plus haut niveau de certification. Chaque certification a un coût et ce coût peut représenter une part importante du budget alloué au développement du système. Nous parlons alors de niveaux de certifications. Dans notre cas, la première machine virtuelle a le plus haut niveau de certification, la seconde machine virtuelle a le second plus haut niveau de certification, et ainsi de suite. Cela permet de réduire les coûts de certification entre chaque machine virtuelle. Nous parlons alors de criticité multiple.

Pour les tâches à l'intérieur des machines virtuelles, nous considérons dans cette thèse des tâches sporadiques. Elles sont réactivées avec un intervalle minimal entre deux activations. Nous nous intéressons plus particulièrement à vérifier le respect

des échéances de ces tâches, exécutées au sein de machines virtuelles dont l'ordonnancement est à déterminer.

1.2 Projet CEOS

Les besoins présentés précédemment ne sont pas uniquement motivés par la recherche théorique, mais correspondent également à une réalité industrielle. En effet, cette thèse s'inscrit dans le contexte du projet CEOS. Ce projet est un projet FUI22 dont le but est de développer un drone autonome de surveillance civile. De nombreux partenaires industriels, tels que la DGAC, Enedis, EDF, les aéroports de Lyon et de Caen Carpiquet, ADCIS, Thalès se sont associés à ESIEE Paris, l'INRIA et l'Université de Lorraine pour le développement de cette solution. Ce drone aura, par exemple, pour mission d'effectuer une inspection pour repérer un défaut sur une clôture ou un défaut sur une ligne électrique. Actuellement, ces inspections sont réalisées de visu et peuvent se révéler très coûteuses (un hélicoptère est parfois déployé pour pouvoir les effectuer).

Au sein de ce consortium, plusieurs équipes, localisées dans des endroits distincts doivent travailler sur les différentes parties du drone. Jusqu'à quatre parties distinctes ont été identifiées : le contrôle-commande, la communication, l'analyse vidéo et la navigation. Ces parties sont développées en parallèle par les différentes équipes. Dans ce contexte, l'utilisation de machines virtuelles se révèle particulièrement adaptée. En effet, chaque équipe peut développer sa partie dans un environnement cloisonné par la machine virtuelle développée et administrée par ses propres soins.

L'entreprise SYSGO, éditrice de l'hyperviseur temps réel PikeOS, est l'un des fournisseurs de technologie du projet CEOS. PikeOS est un hyperviseur certifié avionique ARINC 653, MILS et DO-178C. Il représente donc un candidat de prédilection pour le développement de la partie hypervision du projet CEOS. Étant donné que cette thèse est effectuée en parallèle du projet CEOS, nos travaux porteront sur des modèles basés sur les propriétés de cet hyperviseur. Afin de comprendre, modéliser et travailler sur ces modèles, le début de cette thèse se concentrera sur leur analyse.

Une première remarque peut être faite sur l'hyperviseur PikeOS : l'ordonnancement des machines virtuelles est effectué hors-ligne grâce à une table d'ordonnancement périodique. Il faut donc définir au préalable dans cette table d'ordonnancement quelle machine virtuelle s'exécute et sur quel intervalle de temps. La table d'ordonnancement définie se répète infiniment dans le temps.

C'est notamment ce point qui est novateur, à notre connaissance, dans les recherches effectuées dans le domaine de l'ordonnancement temps réel. Précédemment, les machines virtuelles étaient ordonnancées par des algorithmes précis. Or, dans PikeOS, le choix est laissé libre à l'utilisateur. Il peut donc utiliser des algorithmes existants ou optimiser l'ordonnancement des machines virtuelles par rapport aux tâches contenues dans celles-ci.

1.3 Organisation

Pour répondre au problème de l'ordonnancement des machines virtuelles hypervisées, la thèse est organisée comme suit :

Chapitre 2 : Ce chapitre présente les différents modèles couramment étudiés dans le contexte de l'ordonnancement des systèmes temps réel. Il présente les modèles de tâches, d'architectures matérielles, de contraintes temporelles ainsi que les

algorithmes d'ordonnancements temps réel classiques avec leurs conditions d'ordonnançabilité associées. Nous présentons ensuite le modèle que nous utilisons dans la thèse, à savoir un modèle hiérarchique à deux niveaux dans lequel l'ordonnancement de premier niveau pour les machines virtuelles doit être défini à l'avance par une table d'ordonnancement. Pour cela, nous définissons d'abord des implémentations de systèmes temps réels afin d'avoir une idée des modèles hiérarchiques couramment utilisés. Nous présentons ensuite les différents modèles de criticité et celui que nous utilisons dans cette thèse, pour terminer par une présentation de l'ordonnancement Dual Priority, qui est une classe d'algorithme d'ordonnancement que nous utilisons pour le second niveau.

Chapitre 3 : Dans ce chapitre, nous présentons plus explicitement le modèle hiérarchique utilisé, ainsi que des conditions d'ordonnançabilité aussi bien pour les tâches que pour les machines virtuelles. Pour cela, nous commençons par définir le concept de candidat d'instant critique pour les tâches dans un tel modèle puis deux méthodes de calcul du temps de réponse pire cas des tâches pour ce modèle. La première méthode est spécifique à l'algorithme d'ordonnancement des machines virtuelles utilisé dans le chapitre 4 où une machine virtuelle n'est activée périodiquement qu'une seule fois, et la seconde concerne le cas général d'un motif arbitraire d'activation des machines virtuelles, utilisée dans le chapitre 5.

Chapitre 4 : Ce chapitre permet de définir le concept d'harmonicité des périodes d'activation des machines virtuelles. Cette notion sera utilisée tout au long de la thèse. Elle concerne les périodes des machines virtuelles entre elles. Grâce aux conditions d'ordonnançabilité précédemment définies, nous déduisons les paramètres des différentes machines virtuelles qui sont alors ordonnancées et dont les paramètres sont trouvés de manière itérative, par ordre de criticité décroissant. Pour cela, nous utilisons un algorithme spécifique pour ordonnancer rapidement des machines virtuelles qui ont des périodes harmoniques.

Chapitre 5 : Ce chapitre améliore l'ordonnancement trouvé pour les machines virtuelles dans le chapitre précédent. En effet, cet ordonnancement ne permet pas nécessairement de garantir le respect des échéances de toutes les tâches dans le cas de machines virtuelles avec des périodes harmoniques. Pour compléter l'ordonnancement, nous présentons deux approches basées sur l'ordonnancement P-fair. La première consiste en la résolution d'un problème de programmation linéaire et la seconde en un algorithme à proprement parler. Nous proposons ensuite une solution pour ordonnancer dans un premier temps le plus de machines virtuelles possibles grâce à l'approche basée sur des périodes de machines virtuelles harmoniques, puis de la compléter grâce à un algorithme adapté de P-fair. Cela nous permet d'obtenir un ordonnancement pour le plus de machines virtuelles possibles.

Chapitre 6 : Dans ce chapitre, nous nous intéressons à l'ordonnancement des tâches au sein des machines virtuelles, et non plus à celui des machines virtuelles. Jusqu'à présent, un seul algorithme était utilisé pour l'ordonnancement des tâches, à savoir l'algorithme d'ordonnancement à priorités fixes Rate Monotonic. Les performances en terme d'ordonnançabilité de cet algorithme étant connues pour ne pas être aussi bonnes que pour des ordonnancements à priorité dynamique, nous nous intéressons maintenant à l'ordonnancement Dual Priority. Cet ordonnancement propose un compromis intéressant entre la simplicité d'implémentation de Rate Monotonic dans un système réel et le taux de succès d'un algorithme optimal tel que Earliest Deadline First. Après avoir décrit l'ordonnancement Dual Priority et les différents algorithmes habituellement utilisés pour déterminer les paramètres des tâches spécifiques à cet algorithme, nous intégrons ces derniers à notre modèle hiérarchique à deux niveaux.

Chapitre 7 : Dans ce chapitre, nous présentons les différents outils développés au cours de cette thèse. Nous commençons par introduire notre générateur de tâches aléatoires permettant de stocker les jeux de tâches dans une base de données pour pouvoir y tirer aléatoirement des jeux de tâches correspondant à certains critères (tels qu'une utilisation précise). Nous présentons ensuite un simulateur d'ordonnancement temps réel qui nous permet de tester les algorithmes et d'obtenir des taux de succès. Enfin, nous décrivons un outil qui a été développé dans le cadre du projet CEOS afin de tester les algorithmes dans un projet concret et aussi de configurer l'hyperviseur PikeOS.

Chapitre 8 : Ce chapitre de conclusion rappelle toutes les solutions proposées au cours de ces différentes années de thèse ainsi que les perspectives de recherches qui peuvent être envisagées pour parfaire et continuer les travaux effectués.

Chapitre 2

État de l'art

Dans ce chapitre, nous définissons les notions fondamentales utilisées dans le contexte de l'ordonnancement temps réel. Nous définissons également les différents modèles utilisés dans cette thèse qui nous ont permis d'établir ceux que nous utilisons.

Parmi ceux-là, nous commençons par présenter les différents modèles de tâches, pour savoir ce que nous devons ordonnancer. Nous présentons également les architectures matérielles, celles-ci influant sur la modélisation de la machine pour savoir sous quelles conditions ordonnancer les tâches, avant de présenter différents algorithmes d'ordonnancements, pour avoir un point de vue global de ce qui peut être utilisé et mieux nous situer.

Nous présentons ensuite différentes implémentations de systèmes temps réel, en commençant par les systèmes d'exploitation temps réel, puisque c'est spécifiquement pour de tels systèmes que nos travaux sont développés. Plus précisément, nos travaux s'inscrivent dans le cadre spécifique des systèmes d'exploitation temps réel virtualisés. Nous présentons ainsi ce qu'est la virtualisation et ce qu'elle implique sur les systèmes d'exploitation temps réel.

Du fait que la virtualisation implique deux couches logicielles (l'hyperviseur et le système d'exploitation invité), nous travaillons sur un modèle d'ordonnancement hiérarchique à deux niveaux. Nous présentons, tout d'abord, pour ce modèle une approche issue de l'état de l'art, basée sur des serveurs de tâches, avant de le présenter au travers de la virtualisation.

Chaque système d'exploitation temps réel utilisé dans un système virtualisé prend en charge une application précise. Chaque application n'ayant pas la même criticité, la notion de criticité a été déclinée dans l'état l'art selon deux approches : l'approche à criticité mixte et l'approche à criticité multiple. Nous précisons par la suite quelle approche nous considérons dans cette thèse.

Dans ce manuscrit, nous nous basons sur différentes applications associées à des machines virtuelles. Chaque application est contenue dans une machines virtuelle. Chaque machine virtuelle est associée à au plus un cœur. Ainsi, nous cherchons à savoir comment ordonnancer, dans un contexte multi-cœurs les différentes machines virtuelles ainsi que les tâches contenues dans chaque machine virtuelle. Ce processus peut ainsi se répéter pour les différents cœurs utilisés.

Nous finissons par présenter un dernier algorithme d'ordonnancement, Dual Priority, qui est une approximation sous-optimale d'un ordonnancement à priorités fixes, donnant de très bons taux de succès. Il sera ensuite intégré dans notre modèle, avant de conclure en précisant nos notations et modèles utilisés.

Sommaire

2.1 Modèles pour le temps réel

Dans cette section, nous présentons différents modèles pour l'ordonnancement temps réel. Nous commençons par détailler différents modèles de tâches, pour comprendre ce que nous cherchons à ordonnancer, ainsi que différents modèles d'architecture, pour savoir sur quelle architecture les tâches peuvent être ordonnancées avant de présenter les principaux modèles d'algorithmes d'ordonnancements.

2.1.1 Modèles de tâches

Nous détaillons maintenant les modèles de tâches les plus couramment utilisés dans le contexte de l'ordonnancement temps réel. Une tâche est une modélisation d'un processus informatique qui correspond à un programme à exécuter. Le modèle de tâches le plus élémentaire est le modèle périodique. Celui-ci permet de définir une tâche dont les instances sont activées à intervalles réguliers.

Définition 1 (Jeu de tâches)
Un jeu de tâches est un ensemble de tâches noté τ. Chaque tâche du jeu de tâches est ainsi notée τ_i où i va de 1 à n pour un système à n tâches.

Définition 2 (Instance de tâche)
Une instance de tâche est créée à chaque nouvelle activation de la tâche. C'est donc une succession d'instances de tâches qui sont exécutées par la machine.

Définition 3 (Activation d'une tâche)
L'activation d'une tâche génère une nouvelle instance de tâche qui sera ensuite exécutée en fonction de l'algorithme d'ordonnancement du système.

Définition 4 (Tâche concrète et tâche non concrète)
Une tâche concrète est une tâche dont l'instant de première activation de sa première instance est connu. Sinon, elle est dite non-concrète.

Définition 5 (Jeu de tâches synchrone)
Un jeu de tâches est dit synchrone s'il existe un instant dans la vie du système où toutes les tâches ont une instance activée en même temps.

Nous considérons dans cette thèse des tâches non concrètes (sauf dans le cas particulier du Chapitre 6 sur l'ordonnancement Dual Priority où les tâches sont supposées synchrones et concrètes).

Définition 6 (Tâche périodique)
Une tâche périodique $\tau_i(C_i, T_i, D_i)$ est une tâche qui génère des instances de tâches dont : (i) la durée d'exécution pire cas Worst Case Execution Time (WCET) est notée C_i, l'inter-arrivée entre deux instances successives est égale à T_i et où D_i est l'échéance relative associée à toute instance de la tâche τ_i.

En exemple de tâche périodique, nous pouvons citer la lecture des données recueillies par un capteur. En effet, il est fréquent que ces données soient lues et traitées à intervalles réguliers.

Définition 7 (Tâche apériodique)
Une tâche apériodique est une tâche qui ne s'exécute qu'une fois pendant la durée de vie du système. Contrairement aux tâches périodiques, elle n'est donc définie que par sa durée d'exécution pire cas WCET et son échéance relative. Dans le cas d'une tâche concrète, sa date peut également être précisée.

Définition 8 (Tâche sporadique)
Une tâche sporadique, $\tau_i(C_i, T_i, D_i)$, est une tâche pour laquelle, contrairement à la tâche périodique, T_i est une borne minimale sur l'inter-arrivée entre deux instances successives de la tâche. Les autres paramètres sont identiques au modèle périodique.

Par définition, une tâche sporadique est forcément non concrète. Nous donnons maintenant des notations supplémentaires liés aux tâches périodiques ou sporadiques.

Définition 9 (Utilisation d'une tâche)
L'utilisation U_i de la tâche $\tau_i(C_i, T_i, D_i)$ est définie par $U_i = \frac{C_i}{T_i}$. Elle correspond à la charge induite par la tâche au processeur.

Définition 10 (Utilisation d'un jeu de tâches)
L'utilisation d'un jeu de tâches correspond à la somme des utilisations des tâches. Pour un jeu de tâches τ de n tâches, l'utilisation du jeu de tâches U se définit ainsi :
$U = \sum U_i, \forall \tau_i \in \tau$

La relation entre les échéances relatives des tâches et leur inter-arrivée détermine le modèle d'échéance du jeu de tâches associé. Nous distinguons les modèles à échéance sur requête, à échéance contrainte et à échéance arbitraire.

Définition 11 (Échéance sur requête)
Un jeu de tâches est à échéance sur requête si pour toute tâche τ_i du jeu de tâches, $D_i = T_i$.

Définition 12 (Échéance contrainte)
Un jeu de tâches est à échéance contrainte si pour toute tâche τ_i du jeu de tâches, $D_i \leq T_i$.

Définition 13 (Échéance arbitraire)
Un jeu de tâches est à échéance arbitraire s'il n'existe pas une relation imposée entre T_i et D_i pour toutes les tâches (l'échéance d'une tâche peut être inférieure, égale ou supérieure à l'inter-arrivée de la tâche).

Nous considérons dans cette thèse des jeux de tâches à échéances sur requête.

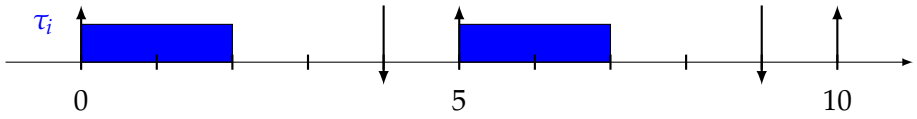


FIGURE 2.1 : Exemple de deux instances de la tâche périodique $\tau_i(2, 5, 4)$

Exemple : La Figure 2.1 indique graphiquement les instants d'exécution de la tâche périodique $\tau_i(2, 5, 4)$. Comme la tâche est seule dans le système, elle s'exécute dès qu'elle est active. Les flèches vers le haut représentent les dates auxquelles la tâche est activée et donc où une nouvelle instance de tâche est créée. Cela correspond à chaque nouvelle période d'activation, toutes les 5 unités de temps, soit en date 0 (date de première activation), 5 et 10. Ces dates coïncident, dans cet exemple, avec les dates d'exécution, la tâche étant seule. Les flèches vers le bas, quant à elles, représentent les dates d'échéances relatives à chaque début de période. Par exemple, la période est de 5, et l'échéance de 4. Les dates d'échéances sont donc en $0 + 4 = 4$ et $5 + 4 = 9$. Les zones pleines correspondent à l'exécution des instance de la tâche. Ici, son WCET est de 2. Son exécution se fait ainsi en pire cas de la date 0 à la date 2 et de la date 5 à la date 7.

Un des plus simples modèles que l'on retrouve dans la littérature de l'état de l'art est celui des tâches périodiques. Il a été initialement défini dans [117]. Les auteurs proposent une analyse du système en considérant des algorithmes d'ordonnancement qui seront vus dans la Section 2.1.3. Ce modèle impose que les tâches soient activées avec une inter-arrivée constante. Cette hypothèse est parfois trop contraignante et peut même s'avérer irréalisable pour certains systèmes qui réagissent à des événements externes au système. C'est pourquoi le modèle de tâche sporadique a été introduit [125]. Ce modèle reprend les mêmes paramètres que le modèle périodique, mais la période devient une période d'inter-arrivée minimale. Cela signifie que la tâche ne peut pas être réactivée avant un certain temps qui correspond à la période d'inter-arrivée minimale.

Pour donner un exemple de tâche sporadique dans un système temps réel, nous pouvons prendre celui des entrées clavier. Si l'utilisateur appuie trop rapidement, les données ne sont plus prises en compte. En revanche, si l'utilisateur appuie avec un intervalle de temps minimum entre deux appuis, les données sont prises en compte. L'utilisateur peut également ne pas appuyer du tout. Le temps entre deux activations de la tâche n'est donc pas connu a priori. L'inter-arrivée est donc une borne inférieure sur les instants d'activation entre deux instances successives de la tâche, il n'y a pas de borne supérieure. La tâche peut ainsi ne jamais se réactiver comme être réactivée à la période d'inter-arrivée minimale suivante.

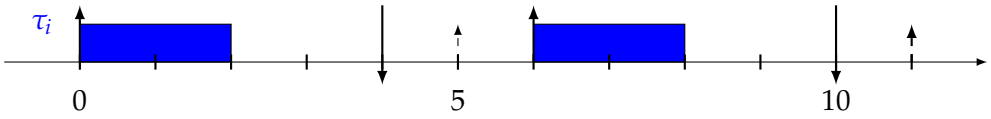


FIGURE 2.2 : Exemple de tâche sporadique $\tau_i(2, 5, 4)$

Exemple : Nous montrons dans la Figure 2.2 une tâche sporadique dans un système où cette tâche est seule. La tâche présentée possède les mêmes paramètres que la tâche de la Figure 2.1. Mais cette fois la tâche est sporadique : sa période devient donc une période d'inter-arrivée minimale. La flèche vers le haut en pointillés indique la période d'inter-arrivée minimale. Ici, la date de la deuxième activation se fait à la date 6, bien que sa période d'inter-arrivée minimale ne soit que de 5. La seconde échéance est ainsi retardée : au lieu de se faire à la date 9, elle se fait ainsi en 10 (seconde date d'activation = 6 + échéance = 4).

Les modèles périodiques et sporadiques sont les modèles fondamentaux puisqu'ils sont utilisés la plupart du temps comme base pour les autres modèles. Cela est dû aux systèmes temps réels. En effet, la plupart des systèmes temps réels sont des systèmes embarquant un grand nombre de capteurs et d'actionneurs, il faut donc pouvoir traiter les informations reçues par les capteurs et agir à intervalles réguliers, ce qui génère donc des modèles périodiques et sporadiques. Nous présentons maintenant une première extension, le modèle périodique avec offset ([73]). Un offset s'applique au modèle de tâches concrètes, il consiste à décaler la date de première activation d'une tâche d'un délai fixé, défini en paramètre de la tâche. Grâce à l'offset, il devient, par exemple, possible de retarder la lecture d'un capteur de façon à empêcher la lecture dès le démarrage du système.

Définition 14 (Offset)
L'offset est un délai imposé pour la date de première activation d'une tâche par rapport à la date de démarrage du système. Ce délai retardera toutes les activations suivantes. La première activation d'une tâche périodique $\tau_i(C_i, T_i, D_i)$ est ainsi faite à O_i, et les activations suivantes à $O_i + k \times T_i$ où $k \in \mathbb{N}$.

Une tâche périodique avec offset s'écrit ainsi :

$$\tau_i(C_i, T_i, D_i, O_i)$$

Un exemple est donné dans la Figure 2.3. Dans cet exemple, l'offset est fixé à 1. La première activation de la tâche ne se fait donc qu'à la date 1, et non plus dès le démarrage du système (à la date 0).

Dans la Figure 2.3, la première instance de la tâche τ_i ne commence donc qu'à la date 1, et la suivante à 6. Son échéance est également décalée aux dates 5 et 10.

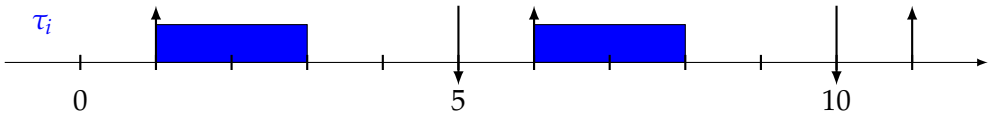


FIGURE 2.3 : Exemple de tâche avec offset $\tau_i(2, 5, 4, 1)$

Définition 15 (Surcoût - Overhead)
Le surcoût est une borne sur le coût système supplémentaire ajouté à l'exécution d'une instance d'une tâche. Il peut intervenir à chaque début d'exécution de l'instance d'une tâche. Ce coût intègre un coût de changement du contexte de l'instance lorsqu'elle est sélectionnée pour être exécutée, auquel s'ajoute un coût de préemption maximum pour l'instance. Le coût de préemption maximum est égal à la somme des coûts de préemption liés aux instances des tâches plus prioritaires qui s'activent pendant la période d'activation de l'instance.

Dans cette thèse, nous ne prenons pas en compte les surcoûts des tâches, ils sont supposés être intégrés aux WCETs des tâches. En revanche, cette notion de surcoût sera par contre prise en compte dans le Chapitre 4, dans le contexte de l'activation de Virtual Machine (VM) par un hyperviseur temps réel.

Maintenant que nous avons vu ce que nous cherchons à ordonnancer, à savoir les tâches, nous décrivons les différents modèles d'architectures en charge de les exécuter.

2.1.2 Modèles d'architecture matérielle

Dans cette section, nous montrons les différents types d'architectures matérielles existants.

Une architecture matérielle est composée d'un ensemble de composants informatiques interconnectés entre eux. Elle contient, au moins, un processeur, de la mémoires Random Access Memory (RAM), des périphériques d'entrée-sortie et de stockages. Un processeur est une entité physique qui regroupe un ou plusieurs cœurs. Un cœur est un ensemble de circuits permettant d'exécuter les différentes instructions qui constituent un programme informatique. Chaque cœur est cadencé par une horloge qui rythme l'exécution des instructions. Plus sa fréquence est élevée, plus le cœur traitera d'informations en une seconde. Notons que les cœurs d'un processeur, et plus précisément ses cœurs, peuvent avoir des architectures différentes. Un processeur, plus précisément ses cœurs, contiennent également différentes mémoires caches. La mémoire cache est une mémoire très proche, voire à l'intérieur du processeur et des cœurs. Elle permet de manipuler des copies d'autres mémoires avec un temps d'accès nettement inférieur aux autres mémoires (le facteur de vitesse d'accès peut dépasser 1000). Il existe différents types de mémoire cache, du plus rapide au plus lent (et de la plus petite mémoire à la plus grande) : L1, L2 et L3. L'ensemble de ces paramètres (type de processeur, fréquence, mémoire cache) induit donc des

temps d'exécution différents pour une même suite d'instructions exécutée sur différents cœurs. Dans une architecture homogène tous les cœurs sont identiques et ont une même fréquence contrairement à une architecture hétérogène.

Toutes ces caractéristiques (fréquence, type de cœur, nombre de cœurs, mémoires caches...) influent très fortement sur la durée d'exécution des programmes. En particulier, si le processeur dispose de plusieurs cœurs, chacun d'eux peut exécuter un programme en parallèle, et donc dans notre cas, plusieurs tâches. Pour une architecture mono-processeur le rôle de l'ordonnanceur pouvait se définir en une seule question : «quand exécuter une tâche ?». Dans le cas multi-processeurs, multi-cœurs, cette question devient double : «quand et où exécuter une tâche ?».

Dans cette thèse, nous supposons que le WCET des tâches est connu. Déterminer la valeur du WCET d'une tâche est un problème de plus en plus complexe avec les architectures matérielles existantes, les mémoires caches introduisent une variabilité des durées d'exécution importante, leur prise en compte est un thème de recherche encore très actif, en particulier avec une approche probabiliste des WCETs ([122]).

Presque tous les processeurs permettent aujourd'hui la virtualisation matérielle. Grâce à cette fonctionnalité plusieurs systèmes d'exploitation peuvent se partager l'ensemble des ressources disponibles comme nous le verrons dans la Section 2.3.2. Cette fonctionnalité récente a un impact très important sur l'ordonnancement et c'est donc sur la prise en compte de ce type d'architecture que reposera cette thèse.

Nous pouvons maintenant nous poser une nouvelle question : «comment ordonnancer les instances de tâches ?». Pour cela, nous proposons d'étudier les différents modèles d'algorithme d'ordonnancement.

2.1.3 Modèles d'algorithme d'ordonnancement

Dans cette section, nous étudions les différents modèles d'algorithme d'ordonnancement, puis, les principaux algorithmes d'ordonnancement, ainsi que les conditions d'ordonnançabilité classiques en contexte mono-processeur.

Nous ne nous intéresserons ici qu'à des algorithmes d'ordonnancement à conservation du travail. C'est-à-dire que si une instance de tâche est active et qu'un cœur du système est libre, celle-ci devra alors être exécutée et son exécution ne pourra pas être retardée.

Définition 16 (Ordonnanceur)
L'ordonnanceur est un service du système d'exploitation en charge de déterminer l'ordre dans lequel les instances des tâches sont exécutées, ainsi que sur quels cœurs.

Définition 17 (Algorithme d'ordonnancement)
Un algorithme d'ordonnancement est un algorithme qui prend en entrée le jeu de tâches avec leurs paramètres et le modèle d'architecture matérielle utilisé, et qui, d'après toutes ces informations produit la liste des instances de tâches à exécuter à tout instant. Il peut se résumer ainsi : «quelle tâche exécuter à quel moment (et où, dans le cas d'une architecture matérielle multi-processeurs) ?».

Nous présentons maintenant les différents modèles d'ordonnancement. Une multitude d'algorithmes d'ordonnancement existe selon le modèle, les contraintes du système, mais également selon les objectifs à atteindre.

Un premier paramètre entre en jeu pour décrire les modèles d'ordonnancement : la préemption. Deux modèles populaires et opposés existent : un ordonnancement préemptif ([117], [113] et [94]) et un ordonnancement non préemptif ([160]).

Définition 18 (Ordonnancement préemptif)
Un ordonnancement préemptif est un ordonnancement pour lequel une instance de tâche en cours d'exécution peut être interrompue par une autre instance de tâche plus prioritaire.

Définition 19 (Ordonnancement non préemptif)
Un ordonnancement non préemptif est un ordonnancement pour lequel si une instance de tâche a commencé son exécution, aucune autre tâche ne peut l'interrompre.

Chaque modèle présente ses avantages et ses inconvénients : le modèle préemptif permet théoriquement d'ordonnancer un plus grand nombre de jeux de tâches (en l'absence de la prise en compte des surcoûts) et des jeux de tâches plus contraints, tandis que le modèle non préemptif permet, comme son nom l'indique, de limiter le nombre de préemptions, et donc de surcoûts des exécutions.

Dans ce manuscrit, nous considérons le problème d'ordonnancement préemptif des tâches au sein d'une VM.

Définition 20 (Ordonnancement partitionné)
Un ordonnancement partitionné prend en compte une architecture multi-processeurs ou multi-cœurs, mais les exécutions des différentes instances de tâches sont restreintes à un seul et unique cœur (pas de migration).

Définition 21 (Ordonnancement global)
Un ordonnancement global est un ordonnancement dans lequel une seule file d'attente globale est utilisée pour sélectionner les instances de tâches à exécuter. Pour une file d'attente de k instances de tâches, les $min(k, m)$ instances de tâches les plus prioritaires y sont ordonnancées, sur les m cœurs d'après leur priorité. La migration entre les cœurs y est autorisée sans restriction.

Une version hybride entre l'ordonnancement partitionné et l'ordonnancement global a également été proposée dans l'état de l'art sous le nom d'ordonnancement semi-partitionné. L'objectif de cet ordonnancement est de mieux maîtriser les coûts de migration des instances de tâches en imposant un schéma de migration statique ([66]).

Il existe plusieurs types d'ordonnancements en fonction des modèles de tâches, de cibles matérielles mais aussi de l'objectif visé : obtenir un ordonnancement non-préemptif ([69]), minimiser les ressources énergétiques ([108], [63] et [78]), minimiser le temps moyen qu'une instance de tâche va mettre pour terminer son exécution. Concernant les cibles matérielles, comme nous l'avons vu dans les modèles d'architecture (Section 2.1.2), il est possible de prendre en compte les caractéristiques des processeurs, comme résumé dans cet état de l'art : [42]. Les ordonnancements d'une architecture multi-cœurs peuvent être partitionnés ([51], [39], [129], [128], [30] et [57]), semi-partitionnés ([8], [6], [28], [168], [99], [99], [100] et [101]) ou globaux ([7], [133], [64], [5], [154], [74], [20], [14] et [15]).

Les principales différences se font sur les possibilités ou non d'exécuter une instance de tâche sur un cœur donné. Par exemple, si le système a besoin d'un algorithme très simple et avec une faible complexité, un ordonnancement partitionné sera en général choisi. Mais l'inconvénient d'un tel algorithme d'ordonnancement, est qu'il peut conduire dans certaines configurations à n'utiliser que 50% des ressources du Central Processing Unit (CPU). Si le système doit pouvoir supporter une utilisation supérieure, un ordonnancement semi-partitionné ou global peut être préférable. Ces deux ordonnancements auront tendance à permettre une plus grande utilisation pour le système mais, en contre-partie, auront tendance à créer plus de

préemptions ou de migrations des tâches. Cela engendrera donc des surcoûts, qui selon le nombre, devront alors être pris en compte.

Lors d'un ordonnancement multi-cœurs, il existe également un effet à prendre en compte : l'effet de Dhall ([51]). Il s'agit d'un effet sur les ordonnancements globaux qui, lorsque l'utilisation du jeu de tâches est élevée, empêche d'ordonnancer le jeu de tâches avec des algorithmes tels que Earliest Deadline First (EDF) ou Rate Monotonic (RM) si le jeu de tâches contient des tâches à grande utilisation et d'autres à faible utilisation.

Dans le contexte de cette thèse, nous ne considérons que les applications définies par un ensemble de tâches à exécuter. Une application est gérée par une VM affectée à un cœur, un cœur pouvant gérer plusieurs VMs.

Nous précisons maintenant les deux types d'ordonnancements utilisés dans la thèse : hors-ligne pour l'ordonnancement des VMs et en-ligne pour l'ordonnancement des tâches au sein des VMs.

Définition 22 (Ordonnancement hors-ligne)
Un ordonnancement hors-ligne est un ordonnancement dans lequel toutes les décisions d'ordonnancement sont prises avant le démarrage du système. L'ordonnancement entier est donc connu à l'avance et peut être paramétré au moyen d'une table d'ordonnancement pour les prises de décision.

Nous montrons dans les Chapitres 4 et 5 comment construire les tables d'ordonnancement des VMs selon deux approches (ordonnancement à périodicité stricte ou basé sur une adaptation de P-fair, algorithme d'ordonnancement initialement conçu pour un ordonnancement multi-processeurs de tâches périodiques). Nous utilisons P-fair pour la construction de la table d'ordonnancement.

Définition 23 (Ordonnancement en-ligne)
Un ordonnancement en-ligne est un ordonnancement dans lequel les décisions d'ordonnancement sont prises par l'ordonnanceur pendant l'activité du système à des instants en fonction de l'activation des instances de tâches. Les décisions d'ordonnancement sont, au moins, prises à l'activation et à la terminaison des instances des tâches (potentiellement plus souvent en fonction de la nature des priorités des instances).

L'invocation de l'ordonnanceur peut être de deux types : événementiel ou à déclenchement temporel (ou time-triggered). Ces deux modes d'invocation se différencient par les instants où l'ordonnanceur est appelé.

Définition 24 (Ordonnanceur à déclenchement temporel (time-triggered))
Un ordonnanceur à déclenchement temporel est un ordonnanceur qui s'active périodiquement avec une période aussi appelée quantum de temps. Ce quantum est paramétrable selon l'ordonnanceur (le plus souvent de l'ordre de 1 ms à 10 ms). Cet ordonnanceur bien que simple peut laisser moins de temps aux tâches pour s'exécuter. En effet, si le quantum de temps est trop réduit, le Real-Time Operating System (RTOS) peut alors passer plus de temps à exécuter l'ordonnanceur que les tâches en elles-mêmes.

Définition 25 (Ordonnanceur événementiel)
Un ordonnanceur événementiel est un ordonnanceur qui est activé à chaque nouvel événement. Pour un ordonnanceur, un événement peut être une activation de tâche ou une fin d'exécution de tâche. Cet ordonnanceur peut être plus efficace et moins gourmand en calculs, mais l'ajout de nouveaux algorithmes prenant en compte d'autres événements que l'activation d'une tâche et la fin d'exécution peut être plus difficile à implémenter.

Nous considérons dans cette thèse un ordonnanceur événementiel pour l'ordonnancement des tâches dans les VMs, car cela correspond à la plupart des RTOSs.

Nous décrivons maintenant les différents modèles d'ordonnancement à priorités fixes issues de l'état de l'art. Cette priorité peut être utilisée hors-ligne pour construire une table d'ordonnancement en simulant un scénario d'activation particulier des tâches (cas synchrone dans le contexte de cette thèse). Dans le cas d'un ordonnancement en-ligne, cette priorité définie pour les instances de tâches actives, est un critère utilisé par l'ordonnanceur pour définir la liste des instances de tâches à exécuter.

Nous introduisons maintenant la notion de complexité d'un algorithme permettant d'expliquer pourquoi certains problèmes d'ordonnancement sont plus complexes que d'autres.

Définition 26 (Complexité d'un algorithme)
Le complexité d'un algorithme permet de mesurer l'efficacité d'un algorithme. Elle exprime le nombre d'opérations à réaliser, en fonction de la taille du problème. Dans notre cas, cela peut par exemple, être le nombre de tâches ou de processeurs. Ces opérations sont alors considérées comme élémentaires à réaliser pour obtenir une décision d'ordonnancement. Une complexité constante s'écrit $\mathcal{O}(1)$, c'est la plus petite complexité possible. Il existe différentes classes de complexité :

- *La classe NP, qui représente la classe des problèmes de décision pour lesquels une solution est vérifiable en temps polynomial.*
- *La classe NP-difficile représente les problèmes qui sont au moins aussi difficiles (dans le sens de la complexité) que les problèmes NP. On peut transformer tout problème de la classe NP en problème NP-difficile par une réduction polynomiale.*
- *La classe NP-complet représente tous les problèmes de décision qui sont à la fois dans la classe NP et la classe NP-difficile. Ces problèmes peuvent se résoudre par une recherche exhaustive, c'est-à-dire en testant toutes les solutions possibles.*

Définition 27 (Priorité fixe au niveau de la tâche)
Une priorité d'une instance de tâche est dite fixe au niveau de la tâche lorsque cette priorité est un paramètre spécifique de la tâche dont hérite l'instance de cette tâche.

Définition 28 (Priorité fixe au niveau de l'instance de tâche)
Une priorité est dite fixe au niveau de l'instance de tâche lorsque cette priorité est un paramètre spécifique attribué à une instance d'une tâche. Deux instances de la même tâche peuvent alors avoir des priorités différentes.

Définition 29 (Priorité dynamique)
Une priorité d'une instance de tâche est dite dynamique lorsque cette priorité est un paramètre spécifique attribué à une instance d'une tâche pouvant évoluer dans le temps.

Le calcul d'une priorité dynamique peut se faire selon des critères tenant compte de l'ordonnancement des autres instances de tâches en cours, contrairement aux priorités fixes.

Pour l'ordonnancement à priorité fixe au niveau des tâches nous avons : [117], [113], [94], [110], [13], [161], [152], [160] et [22]. Pour l'ordonnancement à priorités fixes au niveau de l'instance de tâche : [49], [21], [22], [68], [170] et [171]. Pour l'ordonnancement à priorités dynamiques : [125], [44], [148].

La principale distinction entre l'ordonnancement en-ligne et hors-ligne est le moment où l'ordonnanceur prend les décisions ([60]). Dans le premier cas, toutes les

décisions sont prises selon une table d'ordonnancement construite hors-ligne, avant même que le système ne démarre ([88], [145], [1], [62], [61], [90], [93], [138] et [157]), et dans le second cas, les décisions sont prises à la volée ([49], [105], [125], [89] et [58]). Le premier cas permet donc d'avoir des prises de décision déterministes dont le coût d'exécution est plus simple à maîtriser, puisqu'il suffit de lire la table d'ordonnancement. En revanche, un grand temps de calcul peut être nécessaire pour déterminer la table d'ordonnancement avant la mise en place du système. Le second cas permet, quant à lui, d'avoir un système qui peut prendre en compte des tâches dont la date d'activation n'est pas nécessairement connue a priori.

Nous rappelons maintenant les différents types de conditions d'ordonnançabilité issues de l'état de l'art pour le respect des contraintes temporelles des tâches.

Définition 30 (Condition nécessaire d'ordonnançabilité)
Une condition nécessaire d'ordonnançabilité est une condition qui, si elle n'est pas satisfaite, conduit au non respect des contraintes temporelles des tâches. En revanche, si celle-ci est satisfaite, elle n'est pas suffisante pour conclure si toutes les tâches respecteront leurs échéances.

Définition 31 (Condition suffisante d'ordonnançabilité)
Une condition suffisante d'ordonnançabilité est une condition qui, si elle est satisfaite, garantit le respect des contraintes temporelles des tâches. En revanche, si celle-ci n'est pas satisfaite, il n'est pas possible de déterminer si toutes les tâches respecteront leurs échéances.

Définition 32 (Condition nécessaire et suffisante d'ordonnançabilité)
Une condition nécessaire et suffisante est une condition qui, si elle est satisfaite, permet de garantir que toutes les tâches respecteront leurs contraintes temporelles. Si elle n'est pas satisfaite, elle permet de conclure qu'au moins une tâche ne respectera pas ses contraintes temporelles.

Nous introduisons maintenant la notion d'optimalité d'un algorithme d'ordonnancement.

Définition 33 (Algorithme d'ordonnancement optimal)
Un algorithme d'ordonnancement est dit optimal lorsque l'algorithme trouve une solution pour respecter l'ensemble des contraintes temporelles lorsqu'il en existe une. Si l'algorithme d'ordonnancement optimal ne trouve pas de solution, il n'existe alors aucun algorithme permettant de respecter les contraintes temporelles des tâches (dans la même classe de priorité).

2.2 Algorithmes d'ordonnancement

2.2.1 Algorithmes d'ordonnancement à priorité fixe au niveau des tâches

Notons que l'optimalité des algorithmes d'ordonnancement à priorité fixe au niveau des tâches n'est pas générale mais limitée à cette classe de priorité. Il est en effet possible de trouver des jeux de tâches ordonnançables par des algorithmes à priorité fixe, au niveau des instances de tâches, ou à priorité dynamique alors qu'aucun algorithme d'ordonnancement à priorité fixe au niveau des tâches n'y arrive.

Ordonnancement Rate Monotonic (RM) : [117]

- Avec RM, la priorité d'une instance de tâche est fonction de la période (T_i) de la tâche associée. Plus la période est petite, plus la tâche est prioritaire.

Optimalité :

- RM est optimal pour l'ordonnancement préemptif de tâches sporadiques ou périodiques non concrètes à échéances sur requête .
- RM n'est pas optimal pour des échéances contraintes ou arbitraires.
- RM n'est pas optimal pour l'ordonnancement de tâches périodiques à échéances sur requête avec offset (libre ou non) [95], [73].
- RM n'est pas optimal pour l'ordonnancement de tâches non préemptives [70].

[109] montre qu'en moyenne, la plupart des jeux de tâches à échéance sur requête, avec une utilisation inférieure à 0.88, sont ordonnançables avec RM.

Ordonnancement Deadline Monotonic (DM) : [112]

- Avec Deadline Monotonic (DM), la priorité d'une instance de tâche est fonction de l'échéance relative (D_i) de la tâche associée.
- Plus l'échéance relative est petite, plus la tâche est prioritaire.

Optimalité :

- DM est optimal pour l'ordonnancement préemptif de tâches sporadiques ou périodiques à échéances contraintes. DM n'est plus optimal lorsque les tâches ont des offsets (libres ou non) [95], [73].
- DM n'est pas optimal en contexte non préemptif pour tous les cas d'échéances.

Ordonnancement FP (Priorité fixe) :

- Cet algorithme est utilisé dans le cas d'une attribution arbitraire des priorités. L'instance de tâche la plus prioritaire est sélectionnée en premier.

Optimalité :

- Audsley dans [13] propose un algorithme d'attribution optimal des priorités : valable pour un ordonnancement à priorité fixe dans les cas préemptifs ou non préemptifs, pour des tâches sporadiques ou périodiques non concrètes [13], [70]. Cet algorithme est aussi appelé OPA (Optimal Priority Assignment) dans l'état de l'art. Il est également optimal pour l'ordonnancement de tâches périodiques avec offset (libre ou non) [123], [73].

Principes de l'algorithme :

- Cet algorithme attribue de manière itérative les priorités en partant de la plus basse priorité $(prio = n)$ à la priorité la plus haute $(prio = 1)$.
- La fonction GetFeasible(τ, $prio$) retourne dans l'ensemble courant de tâches τ l'indice de la première tâche ordonnançable sur le niveau de priorité $prio$ ou retourne 0 si aucune tâche n'est ordonnançable. Comme l'algorithme est optimal, il n'existe alors aucune solution d'attribution de priorité permettant de respecter l'ensemble des échéances des tâches dans ce dernier cas. L'algorithme s'appuie sur un calcul de pire temps de réponse d'une tâche pour déterminer si la tâche est ordonnançable.

- On peut choisir arbitrairement une des tâches faisables sans remettre en cause l'optimalité de l'attribution des priorités lorsque plusieurs tâches sont ordonnançables sur un niveau de priorité.

Nous décrivons l'algorithme proposé par Audsley [13] aussi dénommé OPA (Optimal Priority Assignment).

Algorithme 1 : Attribution optimale des priorités OPA

```
1  Taskset τ = {τ_1, ..., τ_n};
2  Integer prio ← n ; Integer j;
3  Boolean failed ← false;
4  tant que failed ≠ true And τ ≠ ∅ faire
5      j=GetFeasible(τ, prio);
6      si j ≠ 0 And failed ≠ true alors
7          SetPriority(j, prio); // Attribution de la priorité prio à la
             tâche τ_j
8          τ = τ − {τ_j}; // τ_j est retirée de l'ensemble τ
9          prio ← prio − 1;
10     sinon
11         failed = true;
```

Nous rappelons maintenant les principales conditions d'ordonnançabilité issues de l'état de l'art pour les ordonnancements à priorité fixe au niveau des tâches dans le cas préemptif, le cas considéré dans cette thèse. Une première propriété importante détermine le scénario pire cas à considérer pour tester l'ordonnançabilité des tâches. Ce scénario correspond au scénario d'activation synchrone des tâches à tester dans le cas d'échéances arbitraires dans la première période d'activité du processeur.

Propriété 1
Le pire temps de réponse d'une tâche τ_i est obtenu pour un ordonnancement préemptif à priorité fixe au niveau des tâches dans la première période active de niveau i du scénario synchrone où seules les instances des tâches de priorité supérieure ou égale à τ_i sont considérées.

[110] montre en contexte préemptif que la plus grande période active de niveau i est obtenue pour une tâche τ_i dans le scénario défini par la Propriété 1.

De plus, il montre que la durée maximale de la période active de niveau i, notée L_i est alors solution de :

$$L_i = \sum_{\tau_j \in hp_i \cup \{\tau_i\}}^{n} \left\lceil \frac{L_i}{T_j} \right\rceil C_j.$$

Dans le cas de l'ordonnancement de n tâches périodiques ou sporadiques non concrètes à échéance sur requête, une condition suffisante (CS) pour le respect des échéances a été proposée par [117] et [50] : $U \leq n(2^{\frac{1}{n}} - 1)$. Cette CS permet de conclure que tout jeu de tâches ayant une utilisation inférieure à $ln(2) > 0.69$ est ordonnançable avec RM. On ne peut par contre pas conclure pour des utilisations supérieures.

Nous rappelons maintenant la condition nécessaire et suffisante (CNS) d'ordonnançabilité en contexte préemptif pour des ordonnancements à priorités fixes au niveau des tâches dans le cas d'échéances arbitraires. Cette CNS est basée sur le calcul

du pire temps de réponse d'une tâche dans le scénario d'activation pire cas, défini par la Propriété 1. Pour ce scénario, le pire temps de réponse d'une tâche τ_i est calculé dans la première période d'activité du processeur de niveau i, où seules deux tâches de priorités supérieures ou égales à τ_i ne sont pas considérées.
Si r_i est le pire temps de réponse d'une tâche τ_i alors une CNS de faisabilité pour un ordonnancement à priorité fixe est : $\forall i \in \{1, \ldots, n\}, r_i \leq D_i$ et $U \leq 1$. Cette CNS est valable en contexte préemptif et non préemptif pour n'importe quelle attribution de priorité fixe.
Condition de faisabilité générale ([110], [160]) :
Cette condition de faisabilité consiste à calculer dans la première période active de niveau i les instants de fin d'exécution successifs $w_{i,q}$ de toutes les instances de τ_i, activée aux instants $q \times T_i$ dans $0, T_i, 2T_i, \ldots \left\lfloor \frac{L_i}{T_i} \right\rfloor$.

Théorème 1
Le pire temps de réponse d'une tâche τ_i ordonnancée par un algorithme à priorité fixe au niveau des tâches est solution de :

$$r_i = \max_{\forall q \in \{0,\ldots,Q\}} (w_{i,q} - q \times T_i) \tag{2.1}$$

Où q est le numéro de l'instance de la tâche τ_i activée en $q \times T_i$ et Q est le nombre maximum d'instances de tâches à considérer pour trouver le temps de réponse pire cas tel que :

$$w_{i,Q} \leq (Q+1) \times T_i \tag{2.2}$$

Et : $w_{i,q}$ est solution de la suite (avec $w^0_{i,q} = qT_i + C_i$) :

$$w^{m+1}_{i,q} = (q+1) \times C_i + \sum_{j \in hp(i)} \left\lceil \frac{w^m_{i,q}}{T_j} \right\rceil \times C_j \tag{2.3}$$

$w_{i,q}$ est l'instant de fin d'exécution de la tâche τ_i activée à l'instant $q \times T_i$ dans la première période active de niveau i. La tâche τ_i activée en $K \times T_i$ se termine avant sa prochaine requête activée en $(K+1)T_i$, elle vérifie $w_{i,K\times T_i} \leq (K+1)T_i$. L'instant $w_{i,K\times T_i}$ est alors la fin de la période active de niveau i pour lequel il n'y a plus d'instances de tâches de priorité supérieure ou égale à τ_i activées avant $w_{i,K\times T_i}$ car s'il y en avait, τ_i activée en $K \times T_i$ ne serait pas terminée en $w_{i,K\times T_i}$. Nous avons donc $w_{i,K\times T_i} = L_i$.

Nous montrons dans la Figure 2.4 un exemple d'ordonnancement RM préemptif.

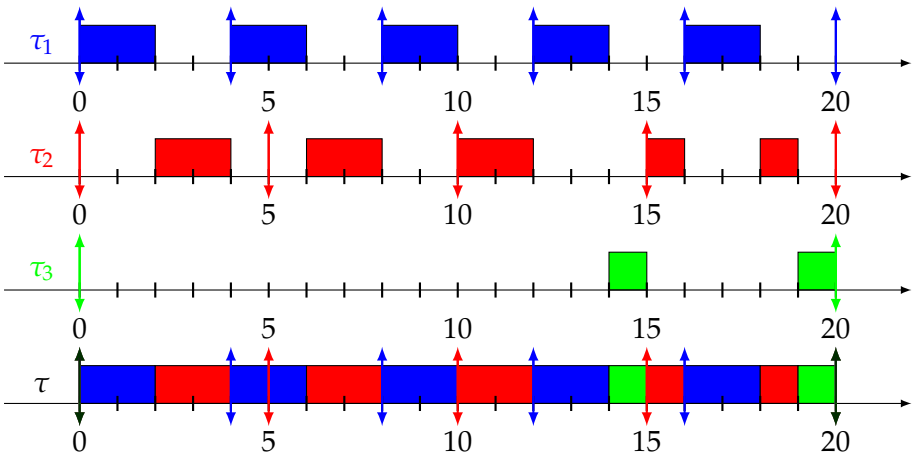


FIGURE 2.4 : Exemple d'ordonnancement RM préemptif avec comme jeu de tâches : τ_1 (2, 4, 4), τ_2 (2, 5, 5) et τ_3 (2, 20, 20)

Les figures se lisent ainsi : nous montrons à la première ligne (en partant du haut) l'ordonnancement de la tâche τ_1 (2, 4, 4), qui a la plus haute priorité, puis sur la seconde ligne τ_2 (2, 5, 5), qui a une priorité moyenne, et enfin sur la troisième ligne τ_3 (2, 20, 20), qui a la plus basse priorité. La dernière ligne correspond à l'ordonnancement de toutes ces tâches sur le cœur qui doit les ordonnancer. Comme déjà expliqué, chaque flèche vers le haut correspond à une activation de la tâche, et chaque flèche vers le bas correspond à une nouvelle échéance absolue de la tâche.

Exemple : La Figure 2.4 illustre l'ordonnancement RM préemptif, la tâche τ_1 s'exécute dès qu'elle est activée, celle-ci pouvant préempter n'importe quelle autre tâche moins prioritaire. Quant à τ_2, elle s'exécute après τ_1, sauf à la date 14, puisque celle-ci a déjà terminé son exécution pour cette activation. Elle reprend donc à la date 15 avant d'être préemptée par τ_1, pour finir son exécution à la date 18. τ_3 s'exécute dans les temps restants, à savoir les dates 14 et 19.

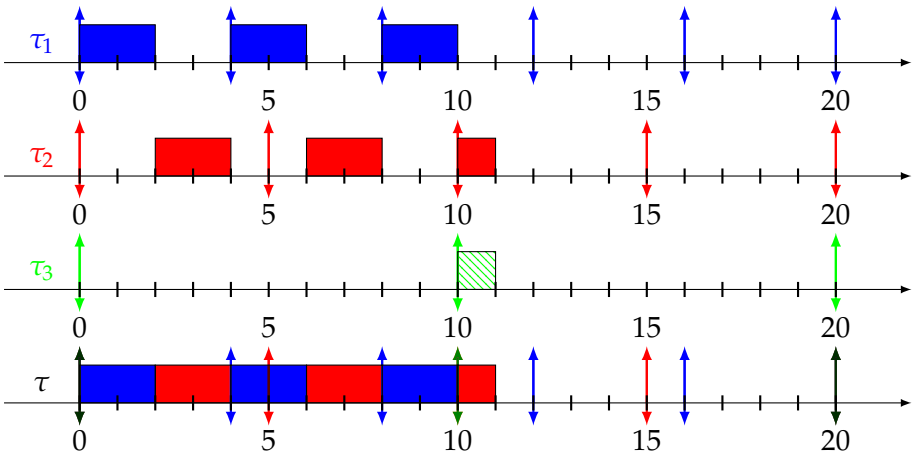


FIGURE 2.5 : Exemple d'ordonnancement RM irréalisable : τ_1 (2, 4, 4), τ_2 (2, 5, 5) et τ_3 (1, 10, 10)

Avec un algorithme d'ordonnancement tel que RM, il n'est pas toujours possible de respecter les échéances des tâches lorsque l'utilisation des tâches est importante (proche de 100%).

Pour illustrer ce phénomène, reprenons le même jeu de tâches que les exemples précédents en modifiant les paramètres de la tâche τ_3, sans pour autant modifier son

utilisation. La tâche $\tau_3(2, 20, 20)$ devient alors $\tau_3(1, 10, 10)$. Celle-ci s'exécute alors moins longtemps, mais plus souvent. Ce simple changement, illustré dans la Figure 2.5, fait qu'à la date 10, aucune exécution de la tâche τ_3 n'a alors pu être faite. Ainsi, son échéance a été dépassée et le système est alors non ordonnançable.

L'ordonnancement RM produit un ordonnancement pour lequel toutes les instances d'une tâche ont la même priorité. Ceci peut limiter l'utilisation maximale du jeu de tâches bien en dessous de 100%.

Cela a motivé, dans l'état de l'art, de considérer des algorithmes d'ordonnancement à priorité fixe au niveau des instances de tâches, qui permettent une utilisation pouvant aller jusqu'à 100% en générant potentiellement des priorités différentes pour les différentes instances d'une même tâche.

2.2.2 Algorithmes d'ordonnancement à priorité fixe au niveau des instances

L'algorithme d'ordonnancement, dans cette classe de priorité, le plus considéré dans l'état de l'art est EDF. La priorité d'une instance de tâche avec cet algorithme est l'échéance absolue de l'instance. EDF attribue la plus grande priorité à l'instance de la tâche d'échéance absolue la plus proche. Lorsque plusieurs instances de tâche ont la même échéance absolue, une règle fixe est utilisée pour ordonnancer les instances ([117], [35], [49], [22], [21], [172], [142] et [98]).

Optimalité :

- EDF est optimal en contexte préemptif pour l'ordonnancement d'un ensemble arbitraire d'instances de tâches sur une architecture mono-processeur [117].

Une condition d'ordonnançabilité pour des tâches périodiques ou sporadiques à échéance sur requête est que l'utilisation du jeu de tâches soit inférieure à 100% ($U \leq 1$).

Exemple : La Figure 2.6 donne un exemple d'ordonnancement EDF en version préemptive.

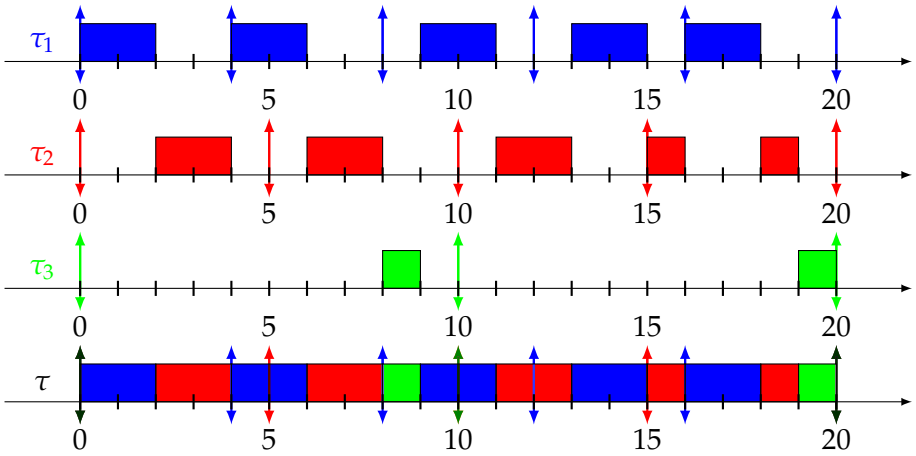


FIGURE 2.6 : Exemple de EDF préemptif avec comme jeu de tâches : τ_1 (2, 4, 4), τ_2 (2, 5, 5) et τ_3 (1, 10, 10)

Graphiquement, l'ordonnancement EDF peut être produit ainsi : l'instance de tâche la plus prioritaire à la date t est l'instance de tâche avec l'échéance absolue la plus proche (représentée par une flèche vers le bas). Par exemple, à la date 0, τ_1 a

la plus grande priorité, suivie de τ_2, elle-même suivie de τ_3. La plus grande priorité avec EDF est donnée à l'instance de tâche avec l'échéance absolue la plus faible. Pour reprendre l'exemple à la date 0, cela donne une priorité de 4 à τ_1, 5 à τ_2 et 10 à τ_3. L'ordonnancement se poursuit ainsi jusqu'à la fin.

Dans le cas préemptif (Figure 2.6), la tâche τ_2 est préemptée par τ_1.

Un problème se pose ici : celui du choix, en cas d'égalité, de la priorité entre plusieurs instances. En effet, à la date 16, toutes les instances de tâches ont alors une priorité de 20. Il peut alors être possible de faire un EDF qui minimisera les préemptions, au lieu de prendre une tâche selon une règle fixe arbitraire ou bien la première tâche possible comme dans l'ordonnancement présenté Figure 2.6. Pour ce faire, il pourrait être intéressant de garder en mémoire la dernière instance de tâche exécutée, et de continuer son exécution lors d'une égalité pour éviter un surcoût d'exécution dû d'un changement de contexte.

2.2.3 Algorithmes d'ordonnancement à priorité dynamique

Un exemple d'algorithme à priorité dynamique, parmi les plus standards est le Least Laxity First (LLF) ([49], [125], [127], [86], [83]). Pour calculer la priorité à chaque instant, cet algorithme utilise le slack, ou la laxité. Cela correspond pour une instance de tâche τ_i activée à l'instant t_i, au temps restant avant la prochaine échéance absolue de l'instance de la tâche à l'instant t moins la durée d'exécution de l'instance de tâche restante. La laxité à l'instant t, $L_i(t)$ d'une instance de tâche τ_i se calcule ainsi :

$L_i(t) = t_i + D_i - t - C_i^{rem}$

Où $t_i + D_i$ est l'échéance absolue de la tâche τ_i, t correspond à l'instant où l'ordonnanceur est appelé, et C_i^{rem} est le temps d'exécution restant de la tâche τ_i à l'instant t. Avec cet algorithme d'ordonnancement, l'instance de tâche avec la plus petite laxité L_i est celle qui sera ordonnancée. Comme pour EDF, en cas d'égalité, une règle fixe sera utilisée. Une bonne façon de procéder est encore une fois, en cas d'égalité, de continuer l'exécution de l'instance de tâche qui était précédemment exécutée, plutôt que d'exécuter une instance de tâche totalement aléatoire. Ainsi, la priorité d'une instance de tâche peut changer à chaque instant.

Optimalité :

- LLF est optimal en contexte préemptif pour l'ordonnancement de tâches périodiques ou sporadiques à échéances arbitraires.

Une condition d'ordonnançabilité pour des tâches périodiques ou sporadiques à échéance sur requête est que l'utilisation du jeu de tâches soit inférieure à 100% ($U \leq 1$).

Un exemple est donné dans la Figure 2.7. Au niveau de l'exécution, cet algorithme d'ordonnancement ressemble à EDF : la première différence, se fait à partir de l'unité de temps 16. En effet, cette fois la tâche τ_2 ne continue pas son exécution et celle-ci est interrompue par τ_1.

Nous montrons, via la Table 2.1, les valeurs de laxité des différentes tâches à partir de l'unité de temps 16, soit à partir du moment où LLF diffère de EDF. Comme indiqué, à partir de l'unité de temps 16, la laxité de τ_1 passe à 2, alors que la laxité de la tâche τ_2 est de 3, celle-ci ayant déjà été exécutée une fois et n'ayant plus qu'une unité de temps à exécuter ($20 - 16 - 1$). La tâche τ_1 devient alors plus prioritaire et préempte la tâche τ_2. À l'unité de temps 17, les trois tâches peuvent être exécutées, celles-ci ayant une laxité de 2. Ici, la tâche τ_1 ayant déjà commencé son exécution, il est alors plus judicieux de la laisser continuer pour qu'elle puisse se terminer.

	$L_i(16)$	$L_i(17)$	$L_i(18)$	$L_i(19)$
τ_1	2	2		
τ_2	3	2	1	0
τ_3	3	2	1	

TABLE 2.1 : Laxité des tâches de la Figure 2.7 à partir de l'unité de temps 16.

Les préemptions sont ainsi minimisées. Un des principaux inconvénients de LLF est qu'il conduit à un grand nombre de préemptions lorsque deux tâches ont la même laxité à un instant donné, ce qui le rend peu compatible avec un ordonnancement de VMs étant donné le coût des changements de contexte dans ce cas.

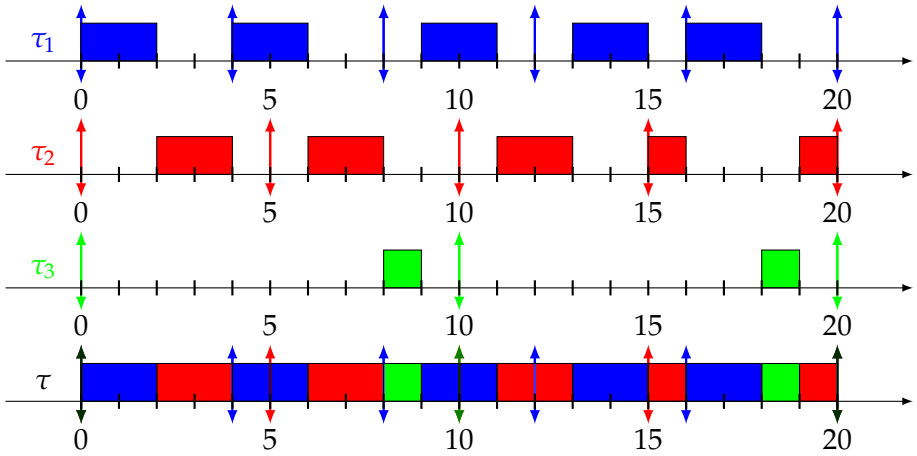


FIGURE 2.7 : Exemple de LLF avec comme jeu de tâches : τ_1 (2, 4, 4), τ_2 (2, 5, 5) et τ_3 (1, 10, 10)

Nous présentons maintenant un dernier algorithme à priorité dynamique : P-fair ([23]). Cet algorithme est dans un premier temps théorique : il définit un ordonnancement fluide pour les tâches en utilisant un quantum de temps minimum infiniment petit. Ainsi, en une seule unité de temps classique, toutes les tâches peuvent avoir été exécutées pendant une proportion de cette unité de temps en fonction de leur utilisation.

Dans cette thèse, une implémentation en-ligne pour un ordonnancement en temps discret sera ensuite proposée, basée sur le concept de lag. Le lag est le retard qu'a pris une instance de tâche en raison de l'ordonnancement en temps discret des instances des tâches. L'objectif de P-Fair est de minimiser le lag des instances des tâches pour reproduire le plus possible un ordonnancement fluide. Cet algorithme sera plus détaillé dans le Chapitre 5 où nous proposons une adaptation hors-ligne de cet algorithme pour construire la table d'ordonnancement de VMs non harmoniques.

Optimalité :

- P-Fair est optimal en contexte préemptif pour l'ordonnancement multi-processeurs (processeurs identiques) pour des tâches périodiques ou sporadiques à échéances sur requête.

Une condition d'ordonnançabilité pour des tâches périodiques ou sporadiques à échéances sur requête est que l'utilisation du jeu de tâches soit inférieure à m

(nombre de processeurs : $U \leq m$) et que sur chaque processeur l'utilisation des tâches affectées au processeur soit inférieure ou égale à 1.

Nous avons maintenant vu les bases essentielles pour l'ordonnancement temps réel. Mais il nous reste à voir ce qu'il en est des systèmes réels, utilisables dans l'industrie.

2.3 Implémentation de systèmes temps réel

Maintenant que les modèles pour l'ordonnancement temps réel ont été présentés dans la Section 2.1, il est important de voir comment sont implémentés les systèmes temps réel. Cette étape nous permet alors de mieux comprendre la thèse et son contexte, celle-ci étant basée sur un modèle extrait d'une implémentation de système temps réel hypervisé, à savoir PikeOS ([158]). Dans un premier temps, nous allons définir comment fonctionnent les systèmes d'exploitation temps réel, puis dans un second temps, expliciter la virtualisation, particulièrement dans le cas du temps réel.

2.3.1 Système d'exploitation temps réel

Définition

Avant de définir ce qu'est un système d'exploitation temps réel, nous proposons de définir un système d'exploitation non temps réel. Pour garder la même terminologie que l'état de l'art, nous utilisons les termes d'Operating System (OS) et Real-Time Operating System (RTOS).

Un Operating System (OS) a plusieurs rôles importants. Il choisit périodiquement la tâche (ou les tâches s'il s'exécute sur une architecture multi-processeurs) à exécuter grâce à son ordonnanceur et sa politique d'ordonnancement. Il est en charge du choix de l'affectation des ressources matérielles pour chacune des tâches. Il offre généralement des services de communications entre les tâches qu'il exécute. Il veille en permanence à l'utilisation correcte des ressources par les tâches exécutées. Il est par exemple capable de détecter et mettre fin à une tâche qui tenterait d'utiliser une zone mémoire non allouée à cette tâche. L'OS a aussi généralement pour rôle d'abstraire l'architecture matérielle grâce à une couche d'abstraction dite HAL (Hardware Abstraction Layer). Grâce à cela les utilisateurs d'un OS n'ont pas nécessairement conscience de ce dont est faite la machine ou de comment celle-ci fonctionne. Pour ce faire, un OS a souvent deux modes de fonctionnement : le mode utilisateur, et le mode noyau (ou mode kernel). Le premier mode, utilisateur, permet à n'importe quel utilisateur d'exécuter des programmes qui demanderont les accès matériels. Le mode noyau, quant à lui, est la partie qui contient les programmes appelés pilotes (drivers en anglais) capables d'accéder directement au matériel.

Un des principaux problèmes pour comparer les OSs est qu'un grand nombre d'entre eux est propriétaire. Cela implique que les seules données connues sont celles que le propriétaire choisit de montrer, ce qui peut grandement compliquer les comparaisons ou même l'utilisation des OSs. En effet, pour prendre l'exemple des pilotes, leur développement peut être ralenti à cause du faible nombre d'informations disponibles sur un OS spécifique. De plus, ceux-ci peuvent également être propriétaires, ce qui complique encore une fois leur portage sur d'autres OSs si besoin. C'est pour cela qu'il est parfois indispensable de faire appel à de l'ingénierie inverse pour porter un pilote propriétaire d'un OS à un autre, si le pilote n'est pas existant sur un autre OS.

Lors de l'utilisation d'un OS, un utilisateur exécute une (ou plusieurs) application(s), qui ont des accès aux ressources, demandées par le développeur de l'application et accordées par l'OS pour ensuite y accéder. Nous pouvons donner en exemple un lecteur de musique. Deux possibilités s'offrent alors : soit les musiques sont stockées directement sur le disque dur de la machine, soit les musiques sont en ligne.

Dans le premier cas, le lecteur de musique devra alors demander l'accès au disque dur à l'OS qui passera alors lui-même par un pilote.

Dans le second cas, le lecteur devra demander un accès au pilote réseau pour pouvoir communiquer avec l'extérieur et ainsi accéder à la musique demandée. En plus de cela, le lecteur de musique devra pouvoir jouer la musique sur, par exemple, des enceintes. Le programme doit alors demander un nouvel accès matériel à l'OS pour pouvoir gérer le son et que la musique soit jouée directement sur les enceintes. Cela se fera donc grâce à un autre pilote.

Du point de vue utilisateur, tout s'est alors passé de manière transparente, sans même avoir besoin de savoir que les musiques sont stockées sur le disque dur ou non. Tout est alors géré par l'OS, les programmes exécutés et les pilotes utilisés. C'est justement ces parties qui seront vues pour définir un RTOS.

Ainsi, l'objectif d'un RTOS n'est plus seulement de permettre d'accéder au matériel sans que l'utilisateur ne le sache, mais également de borner les pires temps de réponse de ces accès. La performance n'est alors plus la principale priorité. Ce sont alors de borner le temps de ces accès, mais également de définir les WCET des tâches qui deviennent essentiels. Dans un OS, la priorité est d'avoir des accès qui ont un bon temps moyen de réponse, mais dans un RTOS, la priorité est d'avoir une borne tout d'abord déterministe, et ensuite la plus petite possible. L'ajout du déterminisme se fait souvent au détriment d'un temps moyen plus grand.

Il existe deux types de temps réel : le temps réel dit souple et le temps réel dit dur. Ces deux types vont énormément influer sur le choix du RTOS. Le temps réel souple consiste à borner de manière déterministe le temps de ces accès, mais une tâche a le droit de dépasser son échéance sans que le système ne soit défaillant. C'est par exemple le cas dans tout ce qui est multimédia. En effet, dans le cas d'un système dont le seul but est de la vidéo, chaque image doit être affichée avant un délai, sans quoi la lecture de la vidéo ne se fait pas. Mais si une image n'a pas eu le temps d'être lue ou calculée, le système peut malgré tout continuer de fonctionner. Seule une image n'aura pas été traitée à temps et sera pas affichée. Si cela ne se produit que rarement, la lecture de la vidéo sera inchangée et l'utilisateur ne pourra pas percevoir la différence.

Cette thèse se place dans un contexte de temps réel dur. Dans ce contexte, il est impératif de garantir qu'aucune tâche ne dépasse jamais son échéance. C'est par exemple le cas sur tous les systèmes de transports : si la tâche permettant, par exemple, de freiner dépasse son échéance, le système devient alors défaillant. C'est ce que nous devons absolument éviter en nous assurant du respect des échéances des tâches du système.

Il est important de connaître le type de RTOS visé et souhaité, puisque sa conception devra changer s'il s'agit de temps réel souple ou dur. Quel que soit le type de temps réel choisi, par rapport à un OS classique, plusieurs points doivent être modifiés pour obtenir un RTOS.

Pour reprendre ce que nous avions déjà vu dans la Section 2.1, un des changements est donc l'ordonnanceur. Pour donner un exemple d'ordonnancement non temps réel, nous pouvons prendre celui par défaut sur un système Linux. Celui-ci est une amélioration de l'algorithme d'ordonnancement Round-Robin ou tourniquet. La tâche sélectionnée pour être exécutée est la tâche qui a la plus grande

priorité. Cette priorité se définit comme suit : à chaque quantum de temps où l'ordonnanceur doit prendre une décision, toutes les tâches actives mais non ordonnancées augmentent leur priorité d'une unité. Ainsi, toutes les tâches sont exécutées au fur et à mesure.

Contrairement aux ordonnanceurs temps réel, cet ordonnanceur ne vise pas à respecter les contraintes temporelles. C'est pour cela que d'autres algorithmes d'ordonnancement doivent être employés. Parmi ceux-là, EDF et RM restent souvent utilisés. Comme EDF est plus difficile à implémenter, il est moins utilisé. Pour des systèmes temps réel avec des contraintes supplémentaires telles que la prise en compte de l'énergie, la minimisation des préemptions, ou bien d'autres approches, ces algorithmes peuvent ne pas suffire. C'est pour pallier cela que des RTOSs proposent de choisir différents algorithmes d'ordonnancement, ou même de créer facilement des algorithmes d'ordonnancement. Cela permet ainsi aux utilisateurs de RTOS d'avoir un ordonnanceur qui répond mieux aux besoins, et qui d'obtenir un plus grand nombre de jeux de tâches ordonnancé.

Dans un RTOS, comme dans un OS classique, les tâches peuvent être dans quatre états distincts : elles sont soit en cours d'exécution, soit prêtes, soit en attente, soit bloquées parce que la tâche est en attente de libération de ressources. Pour faire un parallèle avec les modèles du temps réel, une tâche prête est une tâche qui a été activée, mais qui n'est pas encore en train de s'exécuter. En revanche, une tâche bloquée est une tâche qui a commencé son exécution mais a été interrompue, elle est alors en attente d'un événement, comme par exemple une entrée ou une sortie (cela peut être une simple entrée clavier).

La gestion de ces événements est critique dans un RTOS. Cela créé une interruption au cours de l'exécution de la tâche. Chaque traitement d'interruption est coûteux, et est souvent non borné dans le temps pour les OSs traditionnels. Dans un RTOS, les systèmes sont plus souvent autonomes, les entrées clavier sont donc rares, mais les interruptions logicielles pour attendre le résultat d'une autre tâche, ou les interruptions matérielles de capteurs, sont courantes. Pendant ce temps, une autre tâche peut et doit être exécutée. Chaque interruption va donc créer une préemption, que ce soit par une autre tâche temps réel classique, ou une tâche du RTOS qui devra traiter l'interruption.

Tout comme pour un OS classique, il existe des normes pour créer un RTOS. Ces normes visent à créer une base commune entre tous les RTOSs pour faciliter le passage d'un RTOS à un autre. C'est à ces normes que nous allons nous intéresser maintenant.

Normes

Un RTOS doit donc garantir son fonctionnement, le respect des échéances, borner le traitement des interruptions et s'assurer que les accès aux ressources d'une tâche ne modifient pas le comportement d'une autre tâche. Pour cela, il existe des normes qui sont POSIX.1b, POSIX.1c et POSIX.1d ([53]). Ces normes définissent des Application Programming Interfaces (APIs). Ces APIs sont valables pour Linux. Les normes sont ainsi définies, pour qu'ensuite chaque système souhaitant les implémenter le fasse afin que le code puisse être exporter sur n'importe quel système respectant ces normes. L'idée de ces normes est de définir des standards de programmation internationaux. Cela permet dans un premier temps de ne pas avoir de normes spécifiques à une seule et unique utilisation, mais également de ne pas avoir une entreprise qui va reprendre le monopole de cette norme, celle-ci étant publique.

D'autres normes, spécialisées pour l'automobile existent pour développer des RTOSs. Parmi ces normes, nous pouvons citer Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen / Vehicle Distributed eXecutive (OSEK/VDX) et AUTomotive Open System ARchitecture (AUTOSAR). Ces normes sont à la base faites par un consortium de constructeurs (BMW, Bosch, DaimlerChrysler, Opel, Siemens, et Volkswagen) ensuite rejoint par Renault et PSA. Le but de ce consortium est d'uniformiser la façon de développer des RTOSs dans l'automobile pour que les différents équipements puissent être compatibles, même si les constructeurs n'utilisent pas le même RTOS.

La norme OSEK/VDX est faite pour définir la façon de communiquer, la gestion du réseau, mais également pour gérer certaines parties du RTOS. Cette norme se différencie principalement de la norme POSIX par son développement statique : tous les accès aux périphériques doivent être définis au préalable, les tâches doivent être définies avant, tout comme les sémaphores et certaines parties de la mémoire.

La norme AUTOSAR est, quant à elle, une autre norme basée elle-même sur OSEK/VDX. Ainsi, un RTOS qui utilise OSEK/VDX est compatible avec un RTOS utilisant AUTOSAR.

Par la suite, la norme AUTOSAR a évolué en AUTOSAR Automotive Platform ([65]). Cette norme est principalement faite pour communiquer avec des systèmes non AUTOSAR dans le cas, par exemple, de véhicules connectés. Ainsi, la norme POSIX a également été intégrée dans le système, pour que les communications avec des systèmes POSIX soient simplifiées et réalisables. Chaque OS doit obtenir des certifications pour pouvoir devenir un RTOS, ce qui peut être chronophage et difficile.

Certifications

En plus de ces normes, il faut pouvoir certifier les fonctionnalités d'un RTOS. Cela ne se fait pas directement par le développeur du RTOS, mais par un tiers. Pour le temps réel, il existe de multiples certifications. Parmi celles-ci, la Commission Électronique Internationale (CEI) 61508 ([26]) et son adaptation plus récente spécialisée pour l'automobile, International Organization for Standardization (ISO) 26262 ([131]), ou bien DO-178. La norme CEI peut ainsi être déclinée pour chaque secteur d'activité.

La certification CEI 61508 a été créée pour certifier différents composants électriques, électroniques et surtout électroniques programmables. En effet, avant, les composants n'étaient pas programmables, et n'étaient créés que pour un seul but. La certification ne pouvait alors se faire que sur le composant lui-même. En revanche, avec l'arrivée de l'électronique programmable, les composants doivent être certifiés, ainsi que la programmation appliquée à cet objet.

Ces certifications sont divisées en plusieurs parties : les certifications à proprement parler, mais également des exemples, pour mieux visualiser comment appliquer ces normes ainsi que leur but. Concernant les certifications, elles se divisent en plusieurs niveaux : ces niveaux sont appelés Safety Integrity Levels (SILs). Ils sont classés de 1 à 4 où chaque niveau correspond à une certification plus élevée et donc à un système plus fiable.

Un des principaux problèmes de ces certifications est que le niveau de risque zéro n'existe pas. Le but est alors de s'en approcher le plus possible. Ces certifications permettent de garantir, selon le niveau, une fiabilité du système jusqu'à un certain point. Plus le niveau de certification est haut et moins le système doit être sujet à des failles. La certification du CEI date d'entre 1998 et 2000. Son adaptation pour l'automobile par ISO date, quant à elle, de 2011.

Un système ne pouvant être fiable à 100%, les certifications se font en fonction des types de défaillances et de leurs probabilités de défaillance. Elles peuvent prendre une année pour être réalisées. Les probabilités d'occurrences se classent en plusieurs catégories, de la plus probable à la moins probable :

- Fréquente,
- Probable,
- Occasionnelle,
- Rare,
- Improbable,
- Invraisemblable.

Pour chaque type de défaillance, il existe différents types de conséquences, de la plus critique à la moins critique :

- Catastrophique, qui signifie une perte de plusieurs vies humaines,
- Critique, qui signifie une perte de vie humaine,
- Marginale, qui signifie des blessures majeures à une ou plusieurs personnes,
- Insignifiante, qui signifie des blessures mineures à une ou plusieurs personnes.

C'est à partir de ces différentes classifications de probabilités d'occurrences et de types de défaillance que peuvent être certifiés les systèmes selon un certain SIL.

La certification définie par ISO reprend toutes les certifications et tous les exemples de la certification CEI. Elle ajoute en plus des guides de bonne pratique et des parties informatives pour bien procéder, lors du développement, afin de s'approcher de cette certification.

Ces normes et ces certifications n'étant pas simplement théoriques, nous allons maintenant nous intéresser à plusieurs systèmes d'exploitations pour montrer ce qu'ils ont en commun et les différences qui peuvent exister, afin d'étudier les différentes possibilités de choix.

Exemples de systèmes d'exploitation

Nous allons maintenant donner des exemples de RTOSs, certifiés ou non. Il existe deux façons d'obtenir un RTOS. La première est via un RTOS complet, la deuxième est de modifier le noyau d'un OS pour lui donner des propriétés temps réel et ainsi le transformer en RTOS. C'est par exemple le cas de Xenomai [71].

Définition 34 (Noyau d'OS)
Un noyau de système d'exploitation OS (appelé kernel en anglais), est une couche logicielle qui permet d'accéder aux différentes ressources de la machine. Ainsi, c'est la partie qui fait le lien entre la partie matérielle et la partie logicielle.

Pour comprendre comment peut fonctionner une modification de noyau pour obtenir des propriétés temps réel, nous allons commencer par étudier Xenomai. Ce n'est donc pas un RTOS complet avec son propre noyau, mais une modification du noyau de Linux pour y intégrer des propriétés temps réel. Cela permet d'y ajouter

en plus des APIs pour pouvoir effectuer des appels à des fonctions temps réel et ainsi créer des tâches temps réel.

Les modifications du noyau pour y intégrer Xenomai peuvent se faire de deux façons différentes : Cobalt ou Mercury. Ce sont les noms de deux implémentations différentes de Xenomai. Pour Mercury, un seul noyau est utilisé et la partie temps réel n'est qu'émulée et non directement exécutée. Ainsi, le respect des échéances est plus difficile et les garanties sont difficilement respectables dès que la charge processeur du système est élevée. Cet RTOS est donc adapté au temps réel souple.

Pour Xenomai Cobalt en revanche, deux noyaux sont utilisés. Un premier noyau, classique, s'occupera de toutes les tâches non temps réel et des fonctionnalités habituelles du RTOS. Toute la partie temps réel est, elle par contre, gérée par un nouveau noyau. Ainsi, les interruptions et l'ordonnancement de ces tâches sont gérés par ce noyau.

Puisque deux types de tâches sont en concurrence sur ce système, la gestion se fait à l'aide de priorités : il n'existe qu'une seule bande de priorités pour les deux types de tâches. Ainsi, une tâche temps réel est toujours plus prioritaire qu'une tâche classique. Cela permet de s'assurer que les tâches temps réel respectent leurs échéances.

Au niveau de son fonctionnement, Xenomai reprend un ancien projet, Adeos ([167]). Ce projet était constitué d'un nano-noyau pour gérer les interruptions malgré plusieurs noyaux et possiblement plusieurs OSs. Adeos permet dans ce cas précis de séparer les noyaux en plusieurs parties, appelées domaines. Chaque domaine correspond non pas forcément à un noyau complet, mais au moins à la partie du noyau qui gère les interruptions. Les domaines sont totalement indépendants les uns des autres et n'ont aucune visibilité entre eux. Tout passera donc par Adeos, ce qui permet de centrer toutes les interruptions et les communications pour pouvoir les gérer par une partie externe des différents noyaux, et ainsi conserver les propriétés temps réel.

De plus, des tâches temps réel peuvent nécessiter des fonctionnalités Linux non temps réel. La tâche temps réel devra donc faire appel à une fonction Linux classique. C'est également là qu'interviendra Adeos : son rôle sera alors de promouvoir la priorité de la tâche Linux classique non temps réel pour qu'elle ait la même priorité que la tâche temps réel qui l'a appelée. Ainsi, même si une tâche temps réel utilise des fonctionnalités non temps réel, les priorités sont conservées et seule une tâche temps réel de plus haute priorité pourra alors l'interrompre.

D'autres modifications de Linux peuvent également exister, pour des points spécifiques. C'est par exemple le cas de RTnet ([102]). Cette modification permet de remplacer la partie de Linux qui s'occupe spécifiquement de la partie Ethernet et remplace les protocoles par défaut. Ces protocoles sont ainsi remplacés par d'autres, qui eux, sont temps réel et peuvent avoir un temps de réponse borné pour les communications, ce qui n'est pas le cas par défaut des protocoles Ethernet utilisés dans Linux.

Un des principaux inconvénients d'utiliser des modifications de noyau en plus de l'ajout d'un autre noyau comme dans Xenomai ou Real-Time Application Interface (RTAI) ([118]) est qu'il faut réécrire les applications développées spécialement pour le temps réel. Une autre solution, plus simple, peut permettre d'adapter Linux au temps réel. Celle-ci est le patch PREEMPT_RT ([140]). Contrairement à l'autre approche qui ajoute un noyau, celle-ci n'est qu'un patch de Linux. Globalement les performances sont moins bonnes, et toutes les tâches ne respecteront pas forcément leurs échéances, mais toutes les applications sont directement compatibles, ce qui

peut permettre de les tester très facilement. Bien sûr, il ne reste adapté qu'au temps réel souple.

Pour les RTOSs à proprement parler, le principal problème est que ceux-ci sont souvent propriétaires ou à licences protégées, ce qui limite grandement le nombre de papiers disponibles les décrivant ou les comparant. De plus, les bilans comparatifs de RTOSs sont de plus en plus rares, ceux-ci étant principalement utilisés par les industries.

Définition 35 (Open source)
Un logiciel qualifié de open source est un logiciel dont la licence est libre. C'est-à-dire que son code source est ouvert, que sa distribution est libre, tout comme l'accès au code source, et également la création de travaux dérivés.

Malgré cela, des RTOSs open source subsistent. Pour reprendre un exemple qui est basé sur Linux, et donc open source, nous pouvons évoquer ChronOS ([47]). Ce RTOS utilise Linux et le patch PREEMPT_RT. Il est principalement développé pour des recherches académiques. De ce fait, il existe de multiples algorithmes d'ordonnancement pour effectuer des tests, que ce soit en mono-processeur ou multi-processeurs.

Pour reprendre également les normes et certifications précédemment vues, nous pouvons évoquer ERIKA Enterprise [56]. Ce RTOS est également open source et certifié OSEK/VDX. La dernière version a en plus été développée avec pour objectif d'obtenir plus tard la certification AUTOSAR. Ce RTOS est orienté multi-processeurs mais ne contient que peu d'algorithmes d'ordonnancements différents.

Pour continuer avec des licences open source, nous pouvons évoquer Trampoline Operating System ([25]). Ce RTOS est également développé pour OSEK/VDX et AUTOSAR, sans pour autant avoir officiellement les certifications, en plus d'être compatible avec la norme POSIX. Ce RTOS a principalement été développé dans un but pédagogique : il se veut utilisable par des étudiants pour commencer à comprendre comment créer des tâches tout en ayant une idée de comment s'utilise OSEK/VDX, afin d'être plus rapidement opérationnel dans les industries.

Toujours avec une licence open source, nous pouvons maintenant parler de FreeRTOS ([46]). Un des principaux avantages de ce RTOS est qu'il est disponible pour une grande majorité d'architectures, et donc une grande multitude de processeurs différents. Ce RTOS possède de nombreux avantages : tout d'abord sa gratuité, mais également sa fiabilité. En effet, la plus vieille version encore disponible date de 2004, ce qui en fait un RTOS largement testé. De plus, il se décline en de multiples versions, que ce soit AmazonFreeRTOS ou bien SafeRTOS, qui est une version certifiée de FreeRTOS par les certifications CEI 61508 et ISO 26262.

Pour les autres RTOSs avec une licence qui n'est pas libre, il est possible de citer LynxOS ([147]). Ce RTOS est, quant à lui, entièrement adapté à la norme POSIX. Ainsi, n'importe quelle application compatible est directement compatible dans ce RTOS. D'autres versions de ce RTOS existent également, celles-ci étant plus spécialisées, comme par exemple LynxOS-178 compatible avec la certification pour l'avionique DO-178B ([92]).

Pour citer d'autres RTOSs compatibles avec un grand nombre d'architectures, nous pouvons citer MicroC/OS-II ([107]) et MicroC/OS-III ([106]). La principale différence entre ces deux RTOSs est le nombre de tâches. En effet, MicroC/OS-II ne peut contenir que 255 tâches alors que MicroC/OS-III peut en contenir un nombre non déterminé, uniquement borné par le processeur utilisé et la mémoire.

Un autre RTOS du nom de Nucleus RTOS ([77]) existe également. Celui-ci est certifié pour DO-178C ([141]), CEI 61508, CEI 62304 et ISO 26262. Dans ses principales

fonctionnalités, ce RTOS est compatible avec une interface graphique, mais également avec les normes de connexions sans fils comme le WiFi ([36], [12], [85], [163] et [104]) et le Bluetooth ([124] et [45]).

Un dernier RTOS commercial très répandu dans le monde industriel est VxWorks ([126] et [59]). Ce RTOS d'abord développé dans les années 1980 est un des premiers à avoir un accès réseau temps réel dans les années 1990, et reprend la façon d'implémenter les accès réseaux d'un autre OS, à savoir Berkeley Software Distribution (BSD) ([96] et [155]). En juillet 2019, plus de deux milliards d'appareils utilisent ce RTOS. Il existe différentes versions de ce RTOS telle que VxWorks 653 ([143]) qui est certifié pour ARINC 653 et VxWorks Cert Edition, lui certifié pour DO-178C, CEI 61508 et ISO 26262.

Du fait de toutes ces possibilités de choix d'OSs ou RTOSs adaptés à différents besoins que la construction d'un système peut reposer sur plusieurs d'entre eux. C'est notamment le cas du drone dans le cadre du projet CEOS : chaque partie étant indépendante et développée par des équipes différentes avec des contraintes différentes, il est intéressant que chaque partie puisse être développée sur des OSs ou RTOSs différents, selon les besoins requis par chaque application. C'est pour cela que nous avons besoin de la virtualisation.

2.3.2 Virtualisation

Dans cette section, nous allons introduire le principe de la virtualisation. Elle permet d'utiliser différentes VMs, chacune reposant sur des RTOSs potentiellement différents, au sein d'un même système. Pour cela, nous devons ajouter un composant logiciel intermédiaire : l'hyperviseur que nous allons dans un premier temps définir. En effet, son utilisation est un point clé de la thèse et permet de définir les modèles utilisés. Nous allons ensuite relier les hyperviseurs et le temps réel, pour en voir l'intérêt et les contraintes ajoutées, et ensuite découvrir des certifications nécessaires concernant leur fonctionnement dans des systèmes potentiellement non fiables et finir par des exemples d'hyperviseurs.

Définition

Définition 36 (Machine Virtuelle (VM))
Une première définition peut être donnée par Popek ([135]) : "Une machine virtuelle est considérée comme une copie isolée et efficiente de la machine réelle".

Un hyperviseur peut être également appelé Virtual Machine Manager (VMM) selon le vocabulaire employé. Il permet d'utiliser sur une même architecture matérielle plusieurs OSs différents qui seront alors exécutés dans des VMs. Dans ce cas, ils sont appelés OSs invités, puisqu'ils sont gérés par l'hyperviseur et ne sont plus seuls sur le système. Son rôle est d'abstraire l'architecture matérielle pour que l'OS n'ait ni conscience qu'il est en concurrence avec d'autres, ni qu'il n'a pas un accès direct au matériel. L'abstraction se fait principalement grâce à la gestion des entrées/sorties. En effet, cette partie est habituellement effectuée par l'OS lui-même pour tous les accès matériels, elle sera maintenant gérée par l'hyperviseur, qui lui a tous les accès matériels.

Il existe plusieurs classifications pour différencier les hyperviseurs. La première se fait selon le type :

- Hyperviseur de Type-1. Ils sont également appelés natifs ou *bare-metal*. Les hyperviseurs de ce type s'exécutent directement sur le matériel, sans intermédiaire. De ce fait ils sont souvent présentés comme ayant de meilleures performances temps réel. Il s'agit souvent d'un noyau allégé. Nous verrons que PikeOS en fait partie.

- Hyperviseur de Type-2. Contrairement au type précédent, il a besoin d'un OS pour pouvoir s'exécuter. C'est l'OS hôte qui contiendra alors l'hyperviseur pour virtualiser les différents OSs invités. Cela laisse donc moins de ressources pour les différents OSs invités, mais reste plus simple à mettre en œuvre. Citons par exemple VMWare ou VirtualBox.

Une autre façon de classer les hyperviseurs se fait selon le mode de gestion des OSs invités :

- La full-virtualization, qui permet de ne pas modifier les OSs invités et ainsi de virtualiser n'importe quel OS. En revanche, en plus de dépendre de l'hyperviseur, cette façon de virtualiser n'est pas disponible sur toutes les architectures matérielles. Il faut en effet que le processeur possède des instructions de virtualisation supplémentaires pour pouvoir fonctionner.

- La para-virtualization, est quant à elle disponible sur tous les processeurs. En effet, elle ne requiert aucune instruction processeur supplémentaire. En revanche, elle nécessite que l'OS soit modifié pour que celui-ci puisse être virtualisé. En effet, tous les accès matériels ne doivent plus être directement effectués, mais doivent être redirigés vers l'hyperviseur, qui lui pourra gérer ces accès.

Temps réel et hypervision

L'hypervision impose de prendre en compte de nombreux paramètres : l'OS, mais également l'hyperviseur. En effet, puisque selon certains types d'hypervisions, n'importe quel OS peut être utilisé. Celui-ci doit alors être temps réel pour que le système entier puisse être temps réel, à condition que l'hyperviseur soit également temps réel. Si, ne serait-ce qu'un de ces éléments n'est pas temps réel, le système entier ne peut alors pas être considéré temps réel.

Comme défini précédemment dans la Section 2.3, tous les temps de réponse des composants d'un OS doivent être bornés pour être considérés comme un RTOS. Il en va de même pour l'hyperviseur. Celui-ci doit alors pouvoir gérer les interruptions de manière bornée dans le temps. Tout comme pour un système temps réel qui doit ordonnancer des tâches, l'hyperviseur doit alors ordonnancer des OSs (si le système n'est pas temps réel) ou des RTOSs (si le système est temps réel). Cet ordonnancement doit également être géré de manière précise, que ce soit grâce à une gestion statique pré-calculée, ou bien grâce à des algorithmes d'ordonnancement. Il existe alors deux niveaux d'ordonnancement : un premier pour les RTOSs (ou OSs) et un second au sein des RTOSs (ou OSs) pour les tâches. Ce sont alors ces deux ordonnancements qui devront garantir le respect des échéances des tâches pour que le système entier soit considéré comme temps réel.

Certifications

Pour s'assurer qu'un système réponde bien à ces différentes contraintes temps réel, il existe des certifications, que ce soit au niveau de l'hyperviseur lui-même ou des

OSs. Pour cela, deux standards sont grandement utilisés. ARINC 653 et Multiple Independent Levels of Security (MILS). Ces standards visent à effectuer un partitionnement spatial et temporel des différentes ressources. De tels standards sont donc adaptés pour le temps réel dit dur.

Pour le standard ARINC 653, il est principalement utilisé en avionique. Son but est d'abstraire le matériel du point de vue de l'application. Pour cela, des interfaces de programmations sont fournies. De plus, ces interfaces permettent de facilement utiliser des architectures telles que Integrated Modular Avionics (IMA), qui est un autre standard pour le temps réel dur dans l'avionique. Le standard ARINC 653 définit des règles pour la gestion des partitions (terme défini par le standard) et des processus, mais définit également la manière dont sont gérées les communications.

Le MILS, quant à lui, permet un haut niveau de sécurité basé sur les concepts de séparation et de contrôle des flux d'informations. Il permet également le support de composants fiables et non-fiables. Ce standard se concentre plus sur l'assurance d'une isolation des données et des processus, qui ne doit pas pouvoir être contournée. Ces deux standards ont des objectifs similaires et ne sont pas exclusifs. Un hyperviseur peut par exemple satisfaire les deux.

Exemples d'hyperviseurs

Après avoir vu quels étaient les types d'hyperviseurs et les certifications, il est possible de voir des exemples d'hyperviseurs, selon leur type, leur classement et leurs différentes certifications. Cette partie ne peut pas être exhaustive, du fait de la personnalisation possible de certains hyperviseurs (notamment Xen, Kernel-based Virtual Machine (KVM) et L4). De plus, si l'hyperviseur utilise la full-virtualization, n'importe quel autre OS, temps réel ou non, est utilisable, ce qui décuple le nombre de solutions possibles.

Pour des hyperviseurs répondant à au moins un des deux standards vu précédemment (ARINC 653 ou MILS), il est possible de citer :

- LynuxWorks qui est une couche supplémentaire (appelée LynxSecure, qui est l'hyperviseur) ajoutée à l'OS LynxOS (qui est donc le système hôte), ce qui en fait un hyperviseur de Type-2. Cet hyperviseur supporte ARINC 653 et MILS.
- Wind River ([166]), qui est un hyperviseur de Type-1 pour VxWorks et qui supporte également les deux standards.
- Real-Time Systems GmbH ([139]), hyperviseur de Type-1.
- Tenasys eVM for Windows ([159]).
- National Instruments Real-Time Hypervisor ([137]).
- Open Synergy COQOS ([130]).
- PikeOS de Sysgo ([158]).
- XtratuM ([119]).

XtratuM est un hyperviseur de Type-1. Il a d'abord été conçu pour la sûreté des systèmes embarqués en avionique et a, par la suite, intégré le standard ARINC 653. Cet hyperviseur a l'avantage d'avoir une faible empreinte mémoire et un changement de contexte efficace entre les différentes VMs. Cet hyperviseur utilise la paravirtualization et plusieurs OSs y ont été portés, comme par exemple, FreeOSEK RTOS ([134]).

Pour la plupart des hyperviseurs précédemment cités, seule la para-virtualization est disponible. Une solution utilisant la full-virtualization pour VMWare et VirtualBox est disponible dans [84].

Il faut également savoir que, selon les besoins, d'autres hyperviseurs sont orientés systèmes critiques mais ne répondent pas pour autant aux standards précédemment définis. Parmi eux, nous avons Safety PARtition Kernel (SParK), hyperviseur de Type-1 para-virtualisé. Les OSs invités disponibles sont nC/OS-II et une version personnalisée de stand-alone RT Linux (saRTL).

Concernant les solutions qui utilisent Xen en tant qu'hyperviseur de Type-1, une des difficultés est de connaître toutes les personnalisations effectuées. Il existe par exemple trois ordonnanceurs basiques pour les différentes VMs, et il est possible d'en créer de nouveaux. Ces trois ordonnanceurs basiques sont :

- Borrowed Virtual Time (BVT), qui est maintenant déprécié.
- Simple Earliest Deadline First (SEDF), qui est une modification d'un ordonnanceur EDF. Cet ordonnanceur associe à chaque VM des paramètres similaires au WCET et à la période des tâches. Celui-ci est de type partitionné, ainsi, chaque VM est attachée à un CPU. Pour améliorer la performance de cette solution, différents travaux ont été effectués, comme par exemple [121], qui permet de ne plus perdre le temps non utilisé par une VM. En revanche, cela ajoute comme contrainte que chaque VM ne peut contenir qu'une seule tâche. Cette contrainte étant trop forte, d'autres articles, comme [120], lèvent par la suite cette contrainte, en adaptant un modèle hiérarchique dans Xen.
- Credit scheduler, qui est celui par défaut, fonctionne sur du multi-processeurs et permet une répartition automatique de la charge de travail des différentes VMs sur les CPUs inutilisés. Ainsi, aucun CPU ne reste inactif si une VM peut être ordonnancée.

Une comparaison de ces trois ordonnanceurs est donnée dans [34].

En plus de ces changements d'ordonnanceur il est possible d'améliorer les performances de Xen de différentes manières comme par exemple :

- Une amélioration de la performance des VMs temps réel est obtenue dans [169], en donnant une plus grande priorité aux VMs temps réel.
- Des améliorations pour les performances de la lecture audio ont été faites dans [33], en ordonnançant de manière plus régulière les VMs dédiées à l'audio.
- Des améliorations pour les entrées/sorties ont été faites dans [76].
- Des outils pour contrôler la charge de travail des différentes VMs ont également été ajoutés comme dans [165] et [82].

Il existe également les solutions se basant sur KVM, hyperviseur de Type-2. Étant de Type-2, il nécessite donc un OS pour pouvoir être exécuté : cet OS est Linux. KVM est intégré dans le noyau Linux depuis la version 2.6.20. Cet hyperviseur, contrairement à Xen utilise la full-virtualization et ne nécessite donc aucune modification des OSs invités pour que ceux-ci soient compatibles avec la virtualisation. En revanche, étant exécuté dans l'OS, il faut également que le Linux sur lequel il est installé contienne le patch PREEMPT_RT.

Des améliorations de KVM ont été effectuées en modifiant les ordonnanceurs déjà présents. Par exemple dans [37], les auteurs ajoutent en plus d'un ordonnanceur

EDF pour les VMs, un ordonnanceur Constant Bandwidth Server (CBS). Dans un premier temps les entrées/sorties ne sont pas gérées, puis leur gestion est proposée dans [38]. Dans KVM, la gestion du réseau est également incluse.

Une amélioration majeure de KVM est l'utilisation de REal-time SCHeduler framework (RESCH). RESCH est à la base développé pour Linux, sans obligatoirement KVM. RESCH se décompose en quatre sous-parties : RESCH Core, qui est un ensemble de fonctions pour la gestion de ressources temps réel, RESCH Plugin, qui est un ensemble d'algorithmes d'ordonnancement, RESCH Library qui est un ensemble d'APIs pour l'application utilisateur, et RESCH Bench qui est un ensemble d'outils pour des évaluations comparatives applicables également à d'autres OS temps réels, et donc potentiellement aux autres VMs.

L'intégration de RESCH dans KVM a d'abord été faite dans un cadre monoprocesseur [11], puis a été étendue aux multi-processeurs [10]. Ces ajouts permettent, au final, de modifier plus facilement, et sans toucher aux noyaux, les différents algorithmes d'ordonnancement employés, que ce soit pour les VMs ou pour les tâches au sein d'une VM. La contrainte est ici que les différentes VMs doivent utiliser un noyau Linux, RESCH n'étant disponible que pour ce noyau.

L'hyperviseur L4 est un microkernel étendu pour permettre une virtualisation de Type-1, celui-ci est principalement utilisé dans le cadre de la virtualisation pour la téléphonie mobile. Pour cela, il est souvent utilisé avec un premier OS comme Android ou Windows pour l'interface utilisateur, et d'un autre OS temps réel, pour garantir la performance de tâches avec des contraintes temporelles.

L'approche utilisée dans L4 reste proche de Xen, à la principale différence que L4 est un micro-kernel et non un noyau complet. C'est pour cela que ces approches sont souvent comparées, comme par exemple, dans [91]. Cette comparaison montre que les approches basées sur L4 semblent plus performantes que celles basées sur Xen, avec un plus grand niveau de sécurité. Une autre étude [29] a été faite plus précisément sur l'hyperviseur L4/Fiasco avec L4Linux en tant qu'OS. Cette étude montre que les overheads liés à l'hyperviseur en lui-même restent faibles. En revanche, le système requiert plus de ressources en mémoire cache qu'un simple OS temps réel sans virtualisation.

La dernière solution étudiée sera celle de PikeOS de Sysgo [158], certifié DAL-A. Comme vu précédemment, cet hyperviseur utilise les standards ARINC 653 et MILS. Il peut être considéré comme une approche L4, puisque comme le montre [97], le micro-kernel utilisé à la base de PikeOS est P4. P4 est une ré-implémentation, qui se veut fidèle, de L4. PikeOS a été spécialement conçu pour des systèmes critiques. Il a l'avantage de pouvoir être utilisé en full-virtualization ou en para-virtualization. Étant un hyperviseur propriétaire, tout n'est pas modifiable, ainsi son ordonnancement est résumé dans [162]. Basé sur un ordonnancement hors-ligne, celui-ci a l'avantage de pouvoir être entièrement choisi par l'utilisateur pour le premier niveau, à savoir celui des VMs. Cette solution répond également à un certain nombre de certifications, à savoir RTCA DO-178B/C, EN 50218, CEI 61508, ISO 26262 et CEI 62304, ce qui en fait un hyperviseur sûr et sécurisé, y compris en avionique ou en automobile. C'est pour toutes ces raisons, à savoir toutes les certifications et la possibilité de gérer entièrement les VMs au sein du système, que le projet CEOS a retenu cet hyperviseur pour gérer le drone. C'est à partir de cet hyperviseur que sont basés nos différents modèles, comme le modèle hiérarchique à deux niveaux.

Dans le cadre de cette thèse, nous nous concentrons sur PikeOS pour ses certifications et ses possibilités d'ordonnancement au niveau des VMs.

2.4 Modèle d'ordonnancement hiérarchique à deux niveaux

Dans cette section, nous étudions un modèle d'ordonnancement déduit de l'utilisation d'un hyperviseur, vu dans la Section 2.3.2. Il existe plusieurs versions de ce modèle : l'une utilisant des serveurs qui seront ordonnancés au premier niveau d'ordonnancement. Cette première solution permet de laisser une marge lorsqu'un serveur essaie de demander plus de ressources au système, comme par exemple lorsqu'une tâche apériodique apparaît. Nous étudions ensuite une seconde solution, sans serveur, avec un ordonnancement plus libre, dit arbitraire. Dans ce modèle, nous visons à maximiser l'utilisation donnée au premier niveau, sans laisser de marge, mais en permettant d'ordonnancer un plus grand nombre de systèmes où la liste des tâches est fermée, donc sans tâches apériodiques.

2.4.1 Modèle avec serveur au premier niveau

Ce premier modèle est plus ancien, et ne concerne pas uniquement des modèles hypervisés. Ce type de modèle peut être utilisé plus largement, sans la nécessité de machine virtuelle. En effet, dans ce modèle, il s'agit de serveurs pour le premier niveau d'ordonnancement. Les serveurs, sont malgré tout, similaires à des machines virtuelles : ils contiennent des tâches à ordonnancer et sont ordonnancés selon des algorithmes d'ordonnancement, mais ceux-ci sont tous exécutés sur le même OS ou, dans ce cas précis RTOS. La principale différence est que le type de serveur va définir comment celui-ci est ordonnancé mais également comment celui-ci va ordonnancer les tâches.

Ainsi, ce sont les serveurs qui définissent tout l'ordonnancement : ils s'ordonnancent selon certaines règles pré-définies, que ce soit les serveurs eux-mêmes ou les tâches au sein des serveurs. Un tel type de système est souvent utilisé pour ordonnancer différents types d'applications qui sont alors liées aux serveurs. Cela permet d'avoir des applications temps réel, et des applications non temps réel sur le même système, ce qui, à l'époque de leur étude, était un premier pas pour des systèmes plus ouverts.

Les premiers articles utilisant de tels systèmes parlaient alors de "systèmes ouverts", puisque plusieurs types d'applications pouvaient alors être ordonnancées en concurrence dans les différents serveurs. Un autre avantage des serveurs est de pouvoir garantir le respect des échéances des applications au sein de chaque serveur, serveur par serveur.

Les serveurs avaient déjà été utilisés pour d'autres objectifs : ordonnancer des tâches apériodiques au mieux. Cela se faisait aux moyens de serveurs de tâches apériodiques ou bien de serveurs "polling" ([72] et [48]). Dans un tel système, il existait donc deux types d'éléments : des tâches temps réel périodiques et un serveur. Le serveur est donc en charge d'exécuter les tâches apériodiques.

Pour le serveur de tâches apériodiques, une première proposition était de l'utiliser en tant que serveur exécuté en tâche de fond. Cela signifie que si le serveur est considéré comme une tâche, il a alors un WCET infini, une période infinie et une priorité minimale par rapport à toutes les autres tâches. Ainsi, il est exécuté dès que les tâches temps réel ont terminé leur exécution et donc dès que le système est censé être inactif.

Son fonctionnement est ainsi très simple : les tâches apériodiques à exécuter sont contenues dans une liste et chaque tâche apériodique est traitée dans, par exemple, l'ordre d'arrivée, ou selon des priorités si ces tâches en ont. Ce choix dépendra de l'implémentation choisie et des tâches apériodiques du système.

Une autre implémentation proposée dans ce même papier est le serveur "polling". Cette fois, le serveur est considéré comme une tâche temps réel périodique avec une échéance égale à la période. L'équivalent de son WCET est un budget de temps d'exécution. Ainsi, à chaque nouvelle période, ce budget est entièrement rechargé. Le serveur ne peut s'exécuter qu'une seule fois par période. Lors de son exécution, il va exécuter les différentes tâches apériodiques, selon son budget de temps d'exécution restant. En revanche, si son budget arrive à 0, peu importe si les tâches apériodiques ont terminé leur exécution ou non, elles devront alors attendre la prochaine exécution du serveur. Si aucune tâche apériodique n'est présente lors de l'exécution du serveur, celui-ci met automatiquement son budget à 0 pour se recharger à la prochaine période.

Il faut également savoir que ces deux serveurs étaient proposés dans un contexte où l'algorithme d'ordonnancement pour les tâches (et donc le serveur) ne pouvait être que EDF. Cette contrainte étant forte, puisqu'à l'époque peu d'OSs ou RTOSs proposaient l'algorithme d'ordonnancement EDF, de nouvelles propositions utilisant comme algorithme d'ordonnancement RM ont été faites. De plus, cette fois, le système peut fonctionner avec plusieurs serveurs et non plus un seul, créant un modèle hiérarchique à deux niveaux ([103]). Ce papier définit les serveurs sporadiques déjà définis dans la thèse [151] en plus de tests d'ordonnançabilité pour des applications temps réel ordonnancées dans des serveurs ordonnancés via EDF et RM. Comme indiqué dans le nom, au départ, les serveurs sporadiques étaient utilisés pour gérer les tâches sporadiques d'un système. Ce fonctionnement a ensuite été détourné pour exécuter des tâches périodiques ou sporadiques.

Le fonctionnement du serveur sporadique requiert une priorité, qui dépend de sa période. L'algorithme d'ordonnancement utilisé est RM pour les serveurs, et un budget de temps d'exécution, comme pour le serveur "polling". Cette fois-ci, en revanche, ce budget se recharge quand le serveur est inactif ou que son budget passe à 0. Le budget se recharge alors du montant de la tâche à exécuter par le serveur. Ainsi, il ne manque jamais de budget pour exécuter les tâches qui lui sont attribuées.

Avec un tel modèle, une nouvelle question est soulevée : «comment déterminer les paramètres des serveurs (que ce soit la période ou leur budget) pour que les applications aux seins des serveurs soient ordonnançables ?». Pour répondre à cette question, de nombreux algorithmes ont été proposés ([2], [153], [111] et [72]).

Dans ce contexte, le serveur utilisé est un serveur périodique. Ce serveur peut être vu comme une tâche temps réel classique avec période, et comme équivalent du WCET, une quantité de temps d'exécution. Ainsi, le serveur est ordonnancé exactement comme une tâche temps réel, et ira lui-même ordonnancer les tâches temps réel contenues par le serveur pour que l'application soit exécutée.

Définition 37 (Analyse de temps de réponse)
Une analyse de temps de réponse est un procédé, souvent mathématique, pour déterminer le temps de réponse pire cas d'une tâche. Ainsi, elle correspond au temps qu'aura mis une instance de tâche à terminer son exécution à partir du moment où celle-ci est activée. L'analyse détermine ce pire temps.

Pour commencer cette recherche, il fallait d'abord débuter avec une analyse de temps de réponse pour les tâches au sein des serveurs. En effet, sans cela, il n'est alors possible de déterminer le temps de réponse d'une tâche que par simulation, ce qui reste bien trop chronophage par rapport à une analyse de temps de réponse. Une première analyse de temps de réponse a été proposée dans [144] pour des serveurs périodiques (ou "deferables" [156]). Cette analyse de temps de réponse, considérée comme pessimiste, est par la suite améliorée ([3]).Ainsi, à partir de cette formule de

temps de réponse, il est possible de déterminer les paramètres d'un serveur, comme dans [116]. Des améliorations de ces analyses et des simplifications pour des cas particuliers sont également mises en avant dans [43].

Nous reprendrons le même cheminement au cours de la thèse pour notre propre modèle : nous déterminerons d'abord une analyse de temps de réponse, puis nous déterminerons à partir de cela les facteurs à prendre en compte pour déterminer soit les paramètres soit l'ordonnancement des serveurs.

2.4.2 Modèle sans serveur dit arbitraire

Comme indiqué dans la Section 2.3.2, au cours de cette thèse, nous utilisons la virtualisation. Cette virtualisation impose un modèle hiérarchique à deux niveaux : au premier niveau, des VMs sont ordonnancées, alors qu'au second niveau, ce sont les tâches qui sont ordonnancées au sein des VMs.

Ainsi, comme nous l'avons précisé dans la Section 2.4.1, il existait déjà des modèles hiérarchiques à deux niveaux, mais dans un contexte différent. Nous expliquons ici en quoi le contexte est différent et tous les changements que cela implique.

Premièrement, nous n'allons plus parler de serveurs, mais de VMs. Cette modification peut paraître simple, en n'étudiant que le modèle, mais implique beaucoup de changements. Le premier est comme vu dans la Section 2.3.2 qu'au lieu de simples serveurs contenant chacun des applications, nous avons maintenant des VMs contenant des RTOSs contenant des applications et un jeu de tâches associé.

Ainsi, chaque RTOS peut avoir un algorithme d'ordonnancement différent. De plus, cette fois, l'ordonnancement au premier niveau, donc celui des VMs ne sera plus déterminé par un OS ou RTOS mais par un hyperviseur. Selon l'hyperviseur utilisé, l'algorithme d'ordonnancement utilisé peut totalement varier. Dans notre cas, nous allons même parler d'ordonnancement arbitraire, contrairement aux serveurs qui étaient contraints (avec des budgets, des périodes, des algorithmes précis).

Par arbitraire, nous entendons que l'ordonnancement des VMs doit être précisé à l'avance, mais que celui-ci ne doit pas forcément dépendre d'un algorithme d'ordonnancement et peut être aléatoire. Cela implique qu'il faut déterminer un schéma d'ordonnancement des VMs avant de pouvoir les exécuter. Dans la littérature, à notre connaissance, nous n'avons trouvé qu'un seul travail utilisant une approche similaire ([80]). La différence étant que dans ce papier, l'algorithme d'ordonnancement utilisé dans les VMs est EDF, alors que dans notre modèle, et au sein de la thèse, nous utilisons des algorithmes d'ordonnancement à priorité fixe, que ce soit RM, DM ou basés sur attribution arbitraire des priorités. Ces choix se justifient par les contraintes du projet CEOS. En effet, dans ce projet, lié à la thèse, le choix a été fait d'utiliser PikeOS comme hyperviseur. Ce choix impose ce modèle hiérarchique à deux niveaux pour l'ordonnancement des VMs et des tâches au sein des VMs.

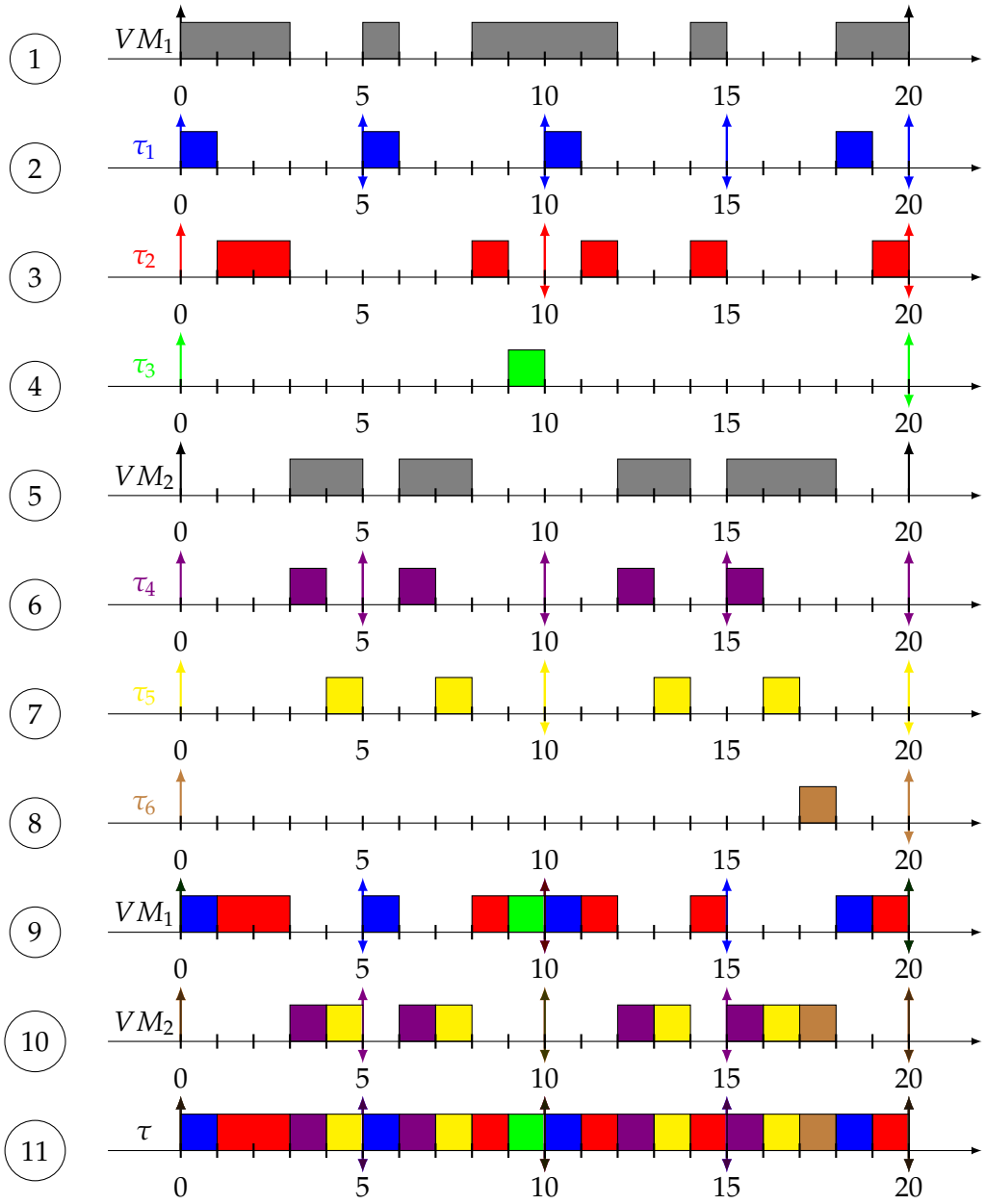


FIGURE 2.8 : Exemple d'ordonnancement arbitraire de deux VMs avec comme jeux de tâches τ_1 (1, 5, 5), τ_2 (3, 10, 10), τ_3 (1, 20, 20) pour VM_1 et τ_4 (1, 5, 5), τ_5 (2, 10, 10) et τ_6 (1, 20, 20) pour VM_2

Nous donnons un exemple d'un tel modèle dans la Figure 2.8. Dans cet exemple, nous prenons un système avec deux VMs : VM_1 et VM_2. Ces deux VMs ont donc un ordonnancement arbitraire, déterminé pour former un exemple. Ces ordonnancements se font ainsi : du temps 0 à 3, puis de 5 à 6, de 8 à 12, de 14 à 15 et enfin de 18 à 20 pour la VM_1, et de 3 à 5, de 6 à 8, de 12 à 14 et de 15 à 18 pour la VM_2. L'ordonnancement des VMs est montré Lignes 1 et 5.

Pour l'ordonnancement des tâches au sein des VMs, nous utilisons ici RM, qui sera grandement utilisé dans la thèse pour des raisons de simplicité et d'efficacité. Dans cet exemple, nous avons deux jeux de tâches différents, un pour chaque VM. Ainsi, les tâches τ_1 (1, 5, 5), τ_2 (3, 10, 10), τ_3 (1, 20, 20) sont associées à la VM_1 et τ_4 (1, 5, 5), τ_5 (2, 10, 10) et τ_6 (1, 20, 20) sont associées à la VM_2.

Étant donné que RM est utilisé, pour le premier jeu de tâches, τ_1 a la plus grande priorité, suivi de τ_2 puis de τ_3. Pour le second jeu de tâches, τ_4 a la plus grande priorité, suivi de τ_5 puis τ_6. Bien évidemment, τ_1, τ_2 et τ_3 ne peuvent s'exécuter que quand la VM_1 est active et donc ordonnancée, idem pour τ_4, τ_5 et τ_6 avec VM_2. Dans un premier temps, nous montrons les ordonnancements obtenus par RM pour chaque tâche isolée (Lignes 2, 3, 4 et 6, 7, 8), puis nous regroupons ces ordonnancements par VM (lignes 9 et 10) pour montrer l'ordonnancement final du système (Ligne 11).

2.5 Criticité

Dans cette section, nous montrons aussi comment la prise en compte de la criticité des tâches permet de réduire le coût de la certification des applications. L'objectif est de pouvoir certifier une application avec le bon niveau d'exigence défini par les contraintes des certifications présentées dans la Section 2.3.1.

Lorsque plusieurs applications de criticité différentes sont exécutées sur la même plate-forme matérielle, l'application de plus haute criticité peut imposer de certifier les autres applications au même niveau d'exigence. Cela est requis par les autorités de certification dans la mesure où les applications peuvent avoir des interférences mutuelles lorsqu'elles partagent les mêmes ressources.

Pour réduire le coût de la certification des applications, il faut pouvoir garantir une bonne maîtrise de ces interférences mutuelles. Deux approches ont été envisagées dans l'état de l'art : l'approche à criticité mixte et l'approche à criticité multiple :

- L'approche à criticité mixte repose sur une isolation temporelle de la criticité. Les tâches sont gérées par le même exécutif temps réel, il faut garantir que l'exécution de tâches de basse criticité impacte de manière maîtrisée les tâches de haute criticité, sans remettre en cause le respect des contraintes temporelles des tâches les plus critiques. Avec l'approche à criticité mixte ([164]), il s'agit d'ajouter un paramètre aux tâches : le niveau de criticité. Avec ce niveau de criticité en plus, il est possible de définir des WCETs par niveau de criticité. Plus le niveau de criticité est élevé, et plus le WCET sera élevé. À tout instant, le système détermine les tâches autorisées à s'exécuter par rapport au niveau de criticité courant du système. Ce niveau est en fonction de la durée d'exécution observée des tâches. Le niveau de criticité augmente lorsque qu'au moins une tâche a une durée d'exécution supérieure au WCET correspondant au niveau de criticité courant. Dans ce cas, toutes les tâches de criticité inférieure ou égale au niveau de criticité courant sont suspendues. Le problème est alors de maîtriser l'impact qu'a eu ces tâches pour pouvoir garantir les contraintes temporelles des tâches qui continuent à s'exécuter. Cela permet ainsi de certifier le système selon chaque niveau de criticité. Cela permet une analyse plus fine du système et de ne plus certifier tout le système au pire cas, ce qui peut s'avérer trop pessimiste. Le problème d'ordonnancement à criticité mixte est un problème NP-difficile ([16] et [19]). Ainsi, nombreux sont les algorithmes d'ordonnancement présentés pour proposer des heuristiques pour l'ordonnancement à criticité mixte. De premiers algorithmes ont ainsi été trouvés pour ce qui concerne l'ordonnancement des instances de tâches. Parmi ces nombreux papiers, il est possible de citer [17], [19], [24], [18], [79], [81], [114], [115], [132], [146], [149] et [150]. Ce type d'ordonnancement n'étant pas vu dans la thèse, nous ne donnerons pas plus de détails.

- L'approche à criticité multiple repose sur une isolation spatiale de la criticité. Les tâches se voient affecter des ressources spécifiques en fonction de leur criticité. Une tâche ne peut donc être impactée temporellement que par des tâches de même criticité.

2.5.1 Notre approche à criticité multiple

Dans notre modèle, nous considérerons que chaque VM a une criticité différente et donc une méthode de certification potentiellement différente. Ici, les VMs sont classés par ordre de criticité décroissant : la plus critique est la première VM. Comme nous utilisons un modèle hiérarchique à deux niveaux (voir Section 2.4), nous appliquons la criticité sur les VMs et non les tâches.

Ici, toutes les tâches au sein d'une même VM sont donc considérées comme ayant le même niveau criticité. La VM la plus prioritaire est ainsi la VM contenant les tâches les plus importantes. Pour reprendre le contexte du drone dans le projet CEOS ([136]), la VM la plus critique contiendra les tâches de l'application contrôle-commande. Ainsi, toutes les parties concernant le contrôle du vol du drone autonome seront implémentées dans cette VM. Le but est donc ici de séparer les différentes fonctionnalités dans les différentes VMs pour que celles-ci soient certifiées au bon niveau de criticité et ainsi réduire les coûts de certification.

Nous utilisons dans cette thèse la seconde approche, donc la criticité multiple. La criticité est définie au niveau d'une application (gérée par une VM) composée de tâches de même criticité. Un hyperviseur de type 1 gère l'accès des VMs aux ressources partagées et garantit par construction l'isolation spatiale de la criticité. Il est ainsi possible de certifier les applications au bon niveau de leur criticité.

L'ordonnancement des VMs tient compte de la criticité. C'est-à-dire que les VMs seront ordonnancées et vérifiées par ordre de criticité décroissante. Ainsi, nous commençons par la première VM, dont nous cherchons l'ordonnancement avant de nous assurer que les tâches respectent leurs échéances. Les autres VMs seront ensuite ordonnancées, par ordre de criticité décroissant. Cela permet de s'assurer que l'ordonnancement d'une VM n'est possible que si les VMs plus critiques ont été ordonnancées. Une priorité d'ordonnancement est ainsi définie : s'il n'est pas possible d'ordonnancer la première VM, qui est la plus critique, alors il est inutile d'essayer d'ordonnancer les VMs suivantes.

2.6 Dual Priority

Dans cette section, nous développons l'ordonnancement Dual Priority ([31] et [41]), en notant ses avantages et ses inconvénients. Pour commencer, nous allons présenter ce qu'est l'ordonnancement Dual Priority.

Ce type d'algorithme a été introduit pour compenser l'utilisation processeur des algorithmes d'ordonnancement à priorité fixe au niveau des tâches. En effet, comme expliqué dans la Section 2.1, ces algorithmes conduisent à des utilisations processeurs inférieures à d'autres algorithmes d'ordonnancement à priorité fixe au niveau des instances de tâches ou à priorité dynamique. Pour situer, nous avions vu qu'un algorithme tel que RM a une condition suffisante explicitant que tout jeu de tâches utilisant moins de 69% de la charge processeur est obligatoirement ordonnançable. Alors qu'un algorithme à priorité fixe au niveau des instances de tâches, tel que EDF est optimal pour une charge processeur inférieure ou égale à 100%.

Des comparaisons entre EDF et Dual Priority et une extension de Dual Priority en un modèle à k priorités sont également données dans [67]. Dans ce papier, il est montré qu'EDF est un cas particulier du modèle à k priorités.

Sachant qu'un ordonnancement à priorité fixe a une implémentation plus simple que l'implémentation d'un ordonnanceur à priorité fixe au niveau des instances de tâches, des recherches sur d'autres classes ont débuté. Ce sont de ces recherches que Dual Priority a émergé. L'idée est simple : la priorité est toujours fixée au niveau des tâches, mais les instances peuvent avoir deux priorités différentes, une à l'activation de l'instance et une autre à une date fixe par rapport à la date d'activation de l'instance.

Le modèle de tâches est ainsi légèrement différent. Une tâche τ_i a maintenant comme paramètres :

$$\tau_i(C_i, T_i, D_i, P_i^1, P_i^2, S_i)$$

Les paramètres C_i, T_i et D_i sont les mêmes que pour les modèles précédents, à savoir C_i est le WCET de la tâche, T_i sa période ou son inter-arrivée minimale et D_i son échéance relative.

En plus de ces paramètres de base, chaque tâche dispose de deux priorités fixes. La première priorité de la tâche τ_i est notée P_i^1 et la seconde P_i^2. Toute instance de la tâche aura alors à son activation une priorité P_i^1.

S_i est une échéance intermédiaire de promotion à partir de laquelle la priorité fixe d'une instance de la tâche passera de P_i^1 à P_i^2. Cette échéance de promotion est contrainte pour une tâche τ_i :

$$0 \leq S_i \leq D_i$$

Lorsque $0 < S_i < D_i$, la tâche a deux priorités associées P_i^1 et P_i^2 différentes telles que $P_i^1 < P_i^2$. Lorsque $S_i = 0$ ou $S_i = D_i$ alors $P_i^1 = P_i^2$.

De plus, toutes les priorités après promotions sont de plus grande priorité que n'importe quelle priorité avant promotion (deux bandes de priorités distinctes).

Pendant longtemps, l'ordonnancement Dual Priority était conjecturé optimal pour des tâches à échéance sur requête (même pour une utilisation processeur de 100%). Il était théoriquement ainsi possible, en fixant les deux priorités fixes et l'échéance de promotion S_i pour une tâche τ_i, de trouver un ordonnancement Dual Priority respectant l'ensemble des échéances tant que leur utilisation totale des tâches était inférieure à 100% ([31] et [40]). Pour autant, aucune preuve n'avait jamais été fournie ([32]) mais aucun contre-exemple n'avait non plus été trouvé. Ce problème est donc resté ouvert pendant 26 ans, jusqu'à ce qu'un contre-exemple soit trouvé dans [54]. Cependant, nous montrons dans le chapitre 6 que le taux de succès de l'ordonnancement Dual Priority reste très élevé (quasi-optimal). Le verrou restant consiste à déterminer les deux niveaux de priorité et l'échéance de promotion pour que l'ordonnancement appliqué respecte les échéances des tâches. Nous présenterons dans le Chapitre 6 différentes méthodes pour parvenir à trouver ces paramètres.

2.7 Résumé des modèles et notations utilisées

L'état de l'art étudié portant sur différents domaines, nous étudions dans cette section un rappel des différentes notations, mais également les parties qui seront utilisées au cours de cette thèse. Pour ce faire, nous reprenons les différentes parties vues

au cours du chapitre de l'état de l'art (Chapitre 2) en spécifiant les points utilisés, en reprenant ou précisant les notations.

Nous avons présenté différents modèles de tâches. Au cours de cette thèse, nous utiliserons principalement le modèle de tâches sporadiques. La notation pour une tâche τ_i est donc :

$$\tau_i(C_i, T_i, D_i)$$

Où C_i est le WCET de la tâche, T_i est la période d'inter-arrivée minimale entre deux activations et D_i l'échéance. De plus, lors des analyses (Chapitre 3), ces tâches seront vues comme des tâches périodiques. La notation restera donc la même, mais la période d'inter-arrivée minimale T_i devient alors une période d'activation, où la tâche s'active alors à chaque période et non plus à au moins une période d'inter-arrivée minimale. Les tâches périodiques sont alors utilisées lors de l'analyse pour obtenir un pire cas : si les tâches sporadiques s'activent à chaque période d'inter-arrivée minimale, la charge processeur est alors plus élevée, et le temps de réponse des différentes tâches est donc plus long.

Nous utiliserons également le modèle de tâches périodiques pour présenter des algorithmes pour les VMs qui lors de leur ordonnancement peuvent être, quant à elles, vues comme des tâches périodiques (Chapitres 4 et 5). Nous présenterons également des algorithmes d'ordonnancement habituellement utilisés sur des tâches périodiques que nous ajouterons dans notre modèle dans le Chapitre 6.

Concernant l'architecture utilisée, nous nous intéresserons ici à des systèmes virtualisés multi-processeurs. Dans ces système, une VM est associée à un cœur, et chaque cœur peut contenir plusieurs VMs. Nous nous intéresserons ensuite à l'ordonnancement des VMs au sein de ses cœurs. Cela revient donc à avoir plusieurs ordonnancements mono-processeurs. Nous ne prenons pas en compte l'influence des caches au cours de cette thèse.

Pour rappel, les modèles de la thèse sont basés sur PikeOS. Cet hyperviseur étant de plus en plus utilisé, les modèles présentés sont donc utilisables sur de vrais systèmes et non pas uniquement théoriques. C'est de cet hyperviseur que nous avons extrait notre modèle hiérarchique à deux niveaux.

Ce modèle est donc totalement flexible : les VMs peuvent être ordonnancées indépendamment en fixant les temps d'exécution à la main. C'est donc le modèle hiérarchique à deux niveaux étudié dans la Section 2.4.2. Ainsi, les VMs ne sont pas bornées à un seul algorithme d'ordonnancement, mais celui-ci peut être entièrement choisi librement. Cet ordonnancement pourrait même également être défini à la main, pour des besoins particuliers. Un premier schéma d'ordonnancement est ainsi défini, et celui-ci se répète dans le temps jusqu'à l'extinction du système. Dans chaque VM est associé un jeu de tâches. Celui-ci ne peut donc pas être ordonnancé et exécuté uniquement lorsque la VM est elle-même active (donc en cours d'exécution). Lorsqu'une autre VM est active, seules les tâches de cette VM peuvent être exécutées. PikeOS permet de certifier l'isolation entre les différentes VMs pour qu'aucune ne puisse influer sur les autres. Cette isolation est spatiale et temporelle : la mémoire de chaque VM est réservée à l'avance, et il n'est pas possible qu'une tâche d'une VM modifie le comportement d'une autre. De plus, lors de la définition des ordonnancements des VMs, aucune ne peut dépasser son temps d'exécution et ainsi influer sur une autre VM.

Pour simplifier les notations et les ordonnancements, les VMs sont vues comme des tâches, mais sans échéance. La VM ρ^i se définit ainsi :

$$\rho^i(C^i, T^i)$$

Où C^i correspond au temps d'exécution disponible pour la VM sur la période T^i. La VM doit donc s'exécuter de C^i unités de temps entre chaque activation toutes les T^i unités de temps. Ce temps d'exécution peut se faire en plusieurs exécutions différentes. Par exemple, il est possible que la VM $\rho^1(2,4)$ s'exécute en 0 et en 3 pour une unité de temps à chaque exécution. Nous prendrons également en compte les overheads des VMs. Ces overheads peuvent être vus de la sorte : le temps de changement de contexte par PikeOS entre chaque VM n'est pas négligeable et est connu à l'avance. La notation de la VM ρ^i devient alors :

$$\rho^i(C^i, T^i, O^i)$$

Où O^i est l'overhead de la VM ρ^i. L'overhead est donc pris en compte à chaque début d'exécution de la VM, et pendant ce temps, aucune tâche ne peut être exécutée : c'est un temps dont l'hyperviseur a besoin pour restituer le contexte de la VM, donc tout ce à quoi la VM a accès, que ce soit en mémoire, ou n'importe quelle autre ressource du système.

Nous ajouterons également une notion de criticité entre les différentes VMs. Cette criticité se fera par ordre des VMs. Ainsi, ρ^1 est la VM la plus critique, ρ^2 est moins critique que ρ^1, mais plus que ρ^3 et ainsi de suite. Le nombre de VMs n'est pas connu à l'avance, les analyses et les algorithmes proposés seront ainsi génériques. Mais la plupart du temps, notamment dans le projet CEOS, le nombre de VM est fixé à quatre. La criticité ne sera utilisée dans cette thèse que pour sélectionner l'ordonnancement des VMs dans un ordre précis. En effet, PikeOS est utilisé dans le projet CEOS pour créer un drone, ainsi s'il n'est pas possible de trouver un ordonnancement pour la première VM qui correspond au contrôle-commande du drone, il ne sert à rien d'essayer d'ordonnancer les autres VMs, le système entier serait alors voué à l'échec.

Chapitre 3

Ordonnançabilité des tâches dans une VM hypervisée : comment définir l'ordonnaçabilité des tâches dans une hiérarchie à deux niveaux ?

Dans ce chapitre, nous expliquons comment garantir l'ordonnançabilité de tâches sporadiques pour un ordonnancement hiérarchique à deux niveaux composé :

- Au premier niveau d'une table d'ordonnancement supposée ici connue.
- D'un ordonnancement de second niveau basé sur un algorithme à priorité fixe préemptif.

Pour ce faire, nous commençons par présenter le modèle que nous utilisons pour caractériser l'ordonnançabilité des tâches, ainsi que des définitions sur l'ordonnancement utilisé. Nous définissons ensuite comment établir le temps de réponse pire cas des tâches. Ce pire temps de réponse est obtenu en caractérisant des candidats d'instant critique à considérer pour obtenir les pires temps de réponse. Nous considérons tout d'abord le cas d'une table d'ordonnancement composée pour une Virtual Machine (VM) d'une seule exécution avec une activation strictement périodique (Section 3.3). Nous considérons ensuite le cas d'une table d'exécution arbitraire.

À partir de ces formules de temps de réponse, nous dérivons pour finir des conditions d'ordonnançabilité pour les tâches associées à chaque VM.

Sommaire

3.1 Contexte

Dans ce chapitre, le modèle étudié est le même que celui présenté dans la Section 2.7. C'est donc un modèle hiérarchique à deux niveaux, où le premier niveau est constitué de VMs auxquelles sont attribuées des jeux de tâches sporadiques. Chaque tâche de chaque VM ne peut donc être ordonnancée que lorsque sa VM associée est elle-même en cours d'exécution. Nous présentons dans ce chapitre des analyses d'ordonnancement de second niveau fonctionnant pour n'importe quel algorithme d'ordonnancement à priorité fixe au niveau des tâches. Cependant, nous nous intéressons plus particulièrement à l'algorithme d'ordonnancement Rate Monotonic (RM), un algorithme d'ordonnancement couramment utilisé dans l'industrie.

Afin de pouvoir analyser notre modèle, nous supposons dans ce chapitre que les paramètres ainsi que l'ordonnancement des VMs sont connus et arbitraires. La table d'ordonnancement des VMs est calculée hors-ligne.

Définition 38 (Ordonnancement strictement périodique)
Un ordonnancement strictement périodique de tâches (respectivement de VMs) est un ordonnancement non préemptif dans lequel le temps entre l'activation des instances de la tâche (respectivement de la VM) et le début de leur exécution est constant et identique pour toutes les instances.

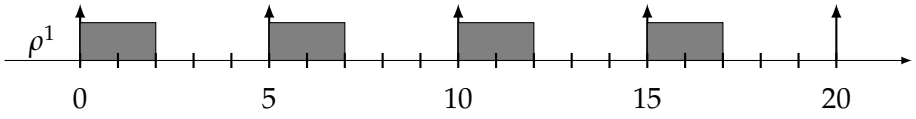


FIGURE 3.1 : Ordonnancement strictement périodique de la VM $\rho^1(2,5)$

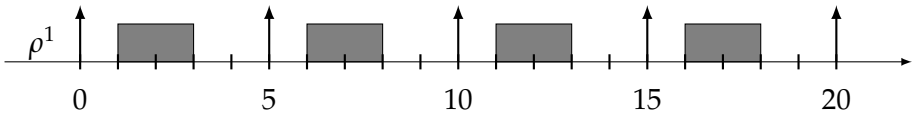


FIGURE 3.2 : Second exemple d'ordonnancement strictement périodique de la VM $\rho^1(2,5)$

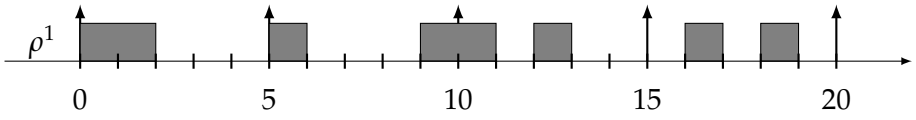


FIGURE 3.3 : Ordonnancement non strictement périodique de la VM $\rho^1(2,5)$

Dans ce chapitre, nous distinguons deux cas : le premier dans lequel les VMs sont ordonnancées de manière strictement périodique, comme dans la Figure 3.1 ou 3.2 et le second dans lequel les VMs ne le sont pas, comme dans la Figure 3.3. Ces distinctions seront notamment utilisées dans la Section 3.3 et 3.4, mais également dans les Chapitres 4 et 5. En effet, grâce à ces distinctions, il sera alors possible de définir deux formules de temps de réponse différentes. La première sera simplifiée grâce à un ordonnancement strictement périodique. La seconde fonctionnera dans le cas général, et il sera également possible de considérer une attribution arbitraire des priorités fixes.

3.2 Concept de candidats d'instant critique

Dans cette section, introduisons le concept de candidat d'instant critique à considérer pour trouver le pire temps de réponse des tâches. Nous allons dans un premier temps définir ce qu'est un candidat d'instant critique. Nous partirons du cas des candidats d'instant critique dans un modèle non hiérarchique, pour, au fur et à mesure, arriver aux candidats d'instant critique dans notre modèle.

Dans le cas de l'ordonnancement préemptif à priorité fixe au niveau des tâches qu'elles soient périodiques (ou sporadiques) sans modèle hiérarchique, [161] et [110] montrent que l'instant critique permet d'obtenir le pire temps de réponse des tâches, en activant de manière synchrone (à un instant 0 de référence) les tâches (premier instant d'activation identique pour toutes les tâches). Ce scénario, permettant d'obtenir le temps de réponse pire cas pour un ordonnancement à priorité fixe au niveau de tâches, est largement utilisé comme base pour tester l'ordonnançabilité d'un jeu de tâches.

Définition 39 (Instant critique)
L'instant critique dans un modèle non hiérarchique avec des tâches sporadiques est le cas d'une activation synchrone de toutes les tâches du système ordonnancées avec la plus petite inter-arrivée possible, donc ordonnancées de manière périodique [161, 110].

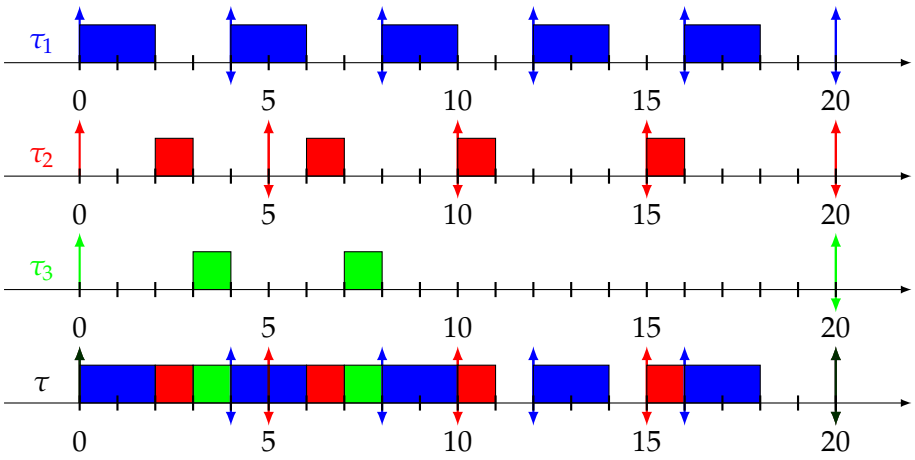


FIGURE 3.4 : Exemple de temps de réponse pire cas de τ_3 en utilisant l'instant critique avec comme jeu de tâches : τ_1 (2, 4, 4), τ_2 (1, 5, 5) et τ_3 (2, 20, 20)

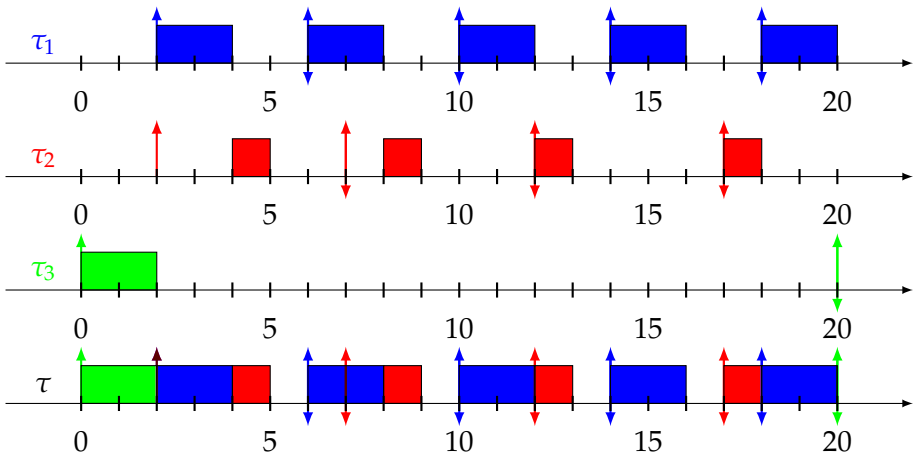


FIGURE 3.5 : Exemple de temps de réponse pire cas de τ_3 sans synchroniser les tâches avec comme jeu de tâches : τ_1 (2, 4, 4), τ_2 (2, 5, 5) et τ_3 (2, 20, 20) où les tâches τ_1 et τ_2 ont un offset de 2

Nous montrons dans les Figures 3.4 et 3.5 des exemples dans lesquels nous comparons les temps de réponse obtenus lorsque les tâches sont activées à l'instant critique et lorsqu'elles ne le sont pas. La première Figure 3.4 montre l'instant critique selon la définition donnée dans [117] et [161], à savoir que toutes les tâches sont synchrones et débutent donc leur activation en même temps. Pour plus de simplicité, nous considérons une activation synchrone des tâches à l'instant 0. Dans ce cas, le temps de réponse de la tâche τ_3 est de 8 comme montré dans la Figure 3.4. En revanche, dans la Figure 3.5, nous donnons un offset de 2 aux tâches τ_1 et τ_2 afin de ne plus activer toutes les tâches au même instant. Cela produit pour la tâche τ_3 un temps de réponse de 2, au lieu de 8 quand toutes les tâches étaient synchrones.

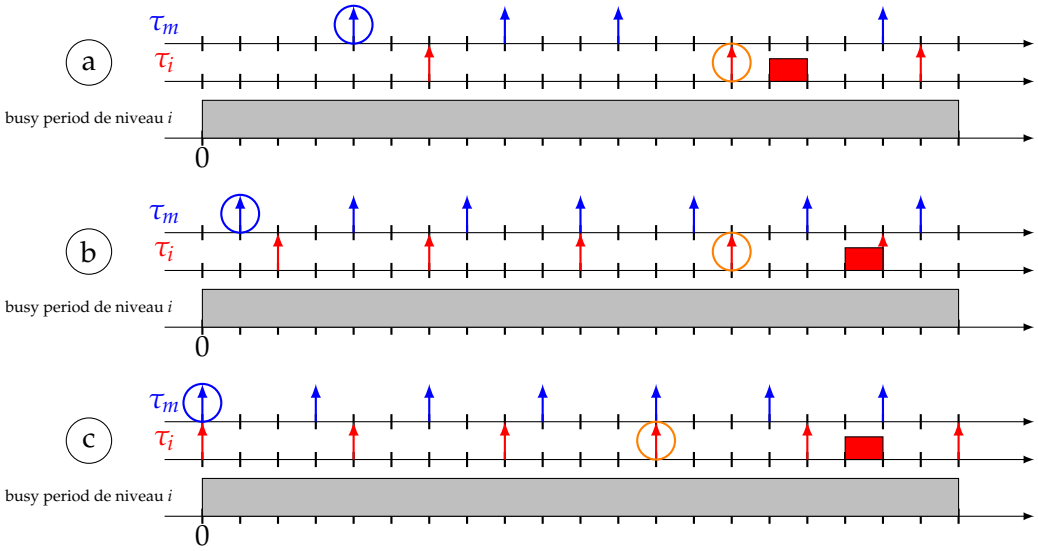


FIGURE 3.6 : Preuve de l'instant critique dans un modèle non hiérarchique avec des tâches à échéance arbitraire

La démonstration de l'instant critique dans un modèle non hiérarchique avec des tâches sporadiques à échéance arbitraire est définie dans [161]. Elle se fait par construction à l'aide de la Figure 3.6 et s'appuie sur la notion de busy period de niveau i.

Définition 40 (Busy period de niveau i)
Une busy period de niveau i est une période d'activité du processeur pendant laquelle le processeur est entièrement occupé à exécuter des tâches de priorité supérieure ou égale à celle de la tâche τ_i.

Principes de la preuve : Prenons un jeu de tâches τ contenant au moins deux tâches : τ_i, qui est la tâche à laquelle on va s'intéresser et τ_m, qui est une tâche de priorité supérieure à τ_i. Dans la Figure 3.6, la busy period de niveau i est représentée par les rectangles grisés. Dans notre cas, nous choisissons l'instant 0. Nous partons d'une tâche quelconque τ_i, de priorité i, pour laquelle nous nous intéressons à une instance de tâche quelconque, pour laquelle nous allons durcir les conditions d'activation pour obtenir l'instant critique.

Nous nous intéressons ici par exemple au comportement de la seconde instance de τ_i dont la date d'activation est ici entourée d'un cercle orange. Comme dit précédemment, nous partons de tâches sporadiques. La première étape est donc de contraindre leur inter-arrivée pour tendre vers un motif d'activation périodique.

Dans le schéma (a), les deux tâches sont sporadiques. La première étape est de les rendre périodiques.

La tâche τ_m est rendue périodique à partir de sa première instance (entourée ici en bleu) pour maximiser le nombre d'instances de τ_m dans la busy period considérée. Ensuite, la tâche τ_i est rendue périodique à partir de l'instance de tâche étudiée, et donc ici de sa seconde instance de tâche.

Ce passage de sporadique à périodique est illustré dans le schéma (b) : nous obtenons maintenant 7 activations de la tâche τ_m au lieu de 4 précédemment et 5 activations de la tâche τ_i au lieu de 3. Le système est ainsi plus chargé. Deux conséquences sont ainsi possibles pour la tâche τ_i :

- Soit le temps de réponse de la tâche τ_i ne change pas.
- Soit le temps de réponse de la tâche τ_i augmente du fait de la surcharge du système et du possible plus grand nombre d'instances de tâches pour chaque tâche après les avoir passées dans un modèle périodique. Cette augmentation est également due à la busy period de niveau i : le système étant entièrement occupé, augmenter sa charge ne peut que dégrader l'état du système.

Dans notre schéma, le durcissement des conditions a augmenté le temps de réponse de l'instance de tâche observée, celui-ci étant passé de 2 à 4.

Après avoir durci le système en passant les tâches du modèle sporadique au modèle périodique, nous cherchons maintenant à les synchroniser. Pour cela, l'activation des premières instances de tâches de τ_i et τ_m passent alors respectivement de l'instant 2 et 1 à 0.

Ce nouveau durcissement des conditions est montré dans le schéma (c) : nous nous intéressons alors toujours à la même instance de tâche de τ_i, entourée en orange. Puisque nous sommes toujours dans la busy period de niveau i, il n'est pas possible que l'instance de tâche observée de τ_i termine son exécution avant son activation non synchrone. En effet, la busy period de niveau i implique qu'une autre tâche de priorité supérieure à la tâche τ_i était exécutée à l'instant 0. Cette tâche ayant une priorité supérieure, activer τ_i à l'instant 0 au lieu de l'instant 2 ne peut pas influer sur la tâche de priorité supérieure.

Seule une des deux implications suivantes est donc possible suite à la synchronisation des tâches à l'instant 0 :

- Soit le temps de réponse de la tâche τ_i ne change pas.

- Soit le temps de réponse de la tâche τ_i augmente.

Dans notre exemple, la date de fin d'exécution de l'instance de tâche de τ_i observée ne change pas. Mais puisque son activation se fait deux unités de temps plus tôt, son temps d'exécution a donc empiré.

Nous avons appliqué ce raisonnement sur uniquement deux tâches : une tâche τ_i et une tâche τ_m de priorité supérieure à τ_i. Nous avons démontré que le temps de réponse était pire ou ne changeait pas lorsque ces deux tâches étaient synchronisées dans un modèle périodique.

Il est possible d'étendre ce raisonnement à toutes les tâches du système pour conclure que l'instant critique de tâches sporadiques avec échéances arbitraires est le cas synchrone où les tâches sont activées de manière périodique.

Notre but est maintenant de montrer que l'activation synchrone est également valable pour notre modèle hiérarchique à deux niveaux. Pour ce faire, nous avons un premier problème : comment prendre en compte le fait que les tâches peuvent être activées à tout moment mais ne peuvent s'exécuter que quand la VM contenant ces tâches est active ? Pour cela, nous définissons, en plus dans le cadre de notre modèle hiérarchique à deux niveaux, le concept de tâche fictive $\overline{\rho^j}$ pour ensuite définir le candidat d'instant critique de la VM.

Définition 41 (Tâche fictive $\overline{\rho^j}$)
La tâche $\overline{\rho^j}$ pour la VM ρ^j est une tâche fictive si elle s'exécute lorsque la VM ρ^j ne s'exécute pas. Elle représente tous les moments d'inactivité de la VM, que ce soit parce que le système exécute une autre VM ou aucune. Cette tâche a donc la plus haute priorité.

Définition 42 (Candidat d'instant critique de la VM ρ^j)
Un candidat d'instant critique de la VM ρ^j correspond à une activation synchrone des tâches de ρ^j avec un début d'une activation de la tâche fictive $\overline{\rho^j}$.

Le pire temps de réponse est ainsi obtenu en considérant le plus grand temps de réponse parmi tous les candidats d'instant critique. En effet, chaque candidat peut correspondre à ce scénario de recherche du pire temps de réponse, mais il n'est pas possible de savoir lequel. Lorsqu'un seul candidat d'instant critique est présent, il est possible de dire directement qu'il s'agit de l'instant critique.

Nous distinguons deux cas possibles pour déterminer les candidats d'instants critiques : le cas le plus simple, dans lequel la VM observée a un ordonnancement strictement périodique, et un second cas général, dans lequel la VM peut avoir un ordonnancement arbitraire. Nous commençons par une démonstration du premier cas.

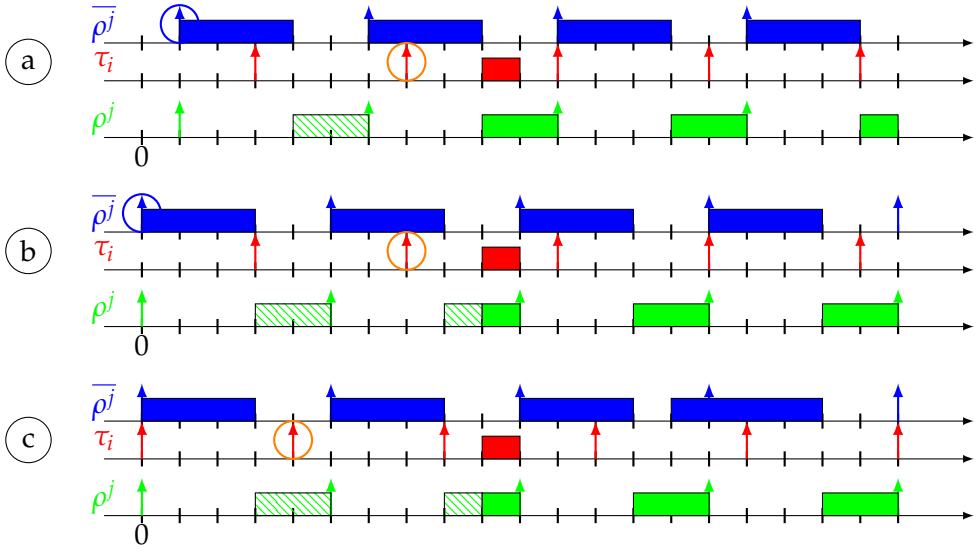


FIGURE 3.7 : Candidats d'instant critique d'une VM strictement périodique

Preuve lors d'un ordonnancement strictement périodique : La démonstration des candidats d'instant critique d'une VM ordonnancée de manière strictement périodique repose sur la démonstration précédente et en reprend ses principes, tout en utilisant la tâche fictive $\overline{\rho^j}$ associée à la VM. En effet, au lieu d'utiliser une tâche quelconque de priorité supérieure, la démonstration se base sur la tâche $\overline{\rho^j}$ utilisée en tant que tâche de priorité supérieure à la place de τ_m. Le raisonnement peut ensuite être appliqué à n'importe quelle autre tâche, de manière à synchroniser leurs activations.

Pour cette démonstration, nous utilisons la Figure 3.7. Nous considérons ici que l'étude est faite dans une busy period de niveau i. Mais elle ne sera pas représentée pour des questions de mise en page. Nous avons donc deux tâches : la tâche étudiée à savoir τ_i, en rouge, et la tâche $\overline{\rho^j}$, en bleu, qui représente donc les moments d'inactivité de la VM. La VM ρ^j est représentée en vert. Rappelons que l'ordonnancement de cette VM est strictement périodique. Tant que cette VM exécute des tâches de priorité supérieure à τ_i, ses exécutions sont hachurées en vert (et sont représentées par des rectangles pleins par la suite, les exécutions ultérieures à celle de τ_i n'étant pas nécessaires pour la démonstration).

Nous partons cette fois directement de tâches périodiques, puisque nous avons déjà démontré que pour étudier le temps de réponse de tâches sporadiques, il est nécessaire de revenir à un modèle périodique. Le schéma (a) représente ainsi les deux tâches τ_i et $\overline{\rho^j}$ qui ne sont pas synchronisées. Nous nous intéressons à nouveau à la seconde instance de la tâche τ_i pour évaluer les conséquences sur son temps de réponse.

Le premier changement effectué est ainsi de débuter l'activation de la tâche $\overline{\rho^j}$ à l'instant 0. Elle est ainsi activée plus tôt. Encore une fois, comme nous sommes dans une busy period de niveau i, les seules conséquences possibles de cette activation plus tôt sont soit que la tâche τ_i est activée au même moment dans le meilleur des cas, soit plus tard. Ce scénario d'activation ne peut pas améliorer le temps de réponse de la tâche τ_i.

Nous synchronisons ensuite la tâche τ_i à la tâche $\overline{\rho^j}$, à l'instant 0. Le résultat est montré dans le schéma (c). Cela implique donc d'avoir plus d'activations de la

tâche τ_i dans le temps d'étude. Nous passons ici de 5 à 6 activations de la tâche. Le système est donc plus chargé. De plus, comme l'étude est toujours effectuée dans la busy period, il n'est pas possible que l'instance de tâche étudiée puisse s'exécuter plus tôt, la VM exécutant déjà des tâches de priorité supérieure avant. Son activation est donc faite plus tôt, mais sa fin d'exécution est plus tardive.

Dans notre cas, le temps de réponse de l'instance de tâche étudiée est donc plus long. La synchronisation entre $\overline{\rho^j}$ et τ_i nous permet donc de trouver un temps de réponse plus long. Encore une fois, en étendant ce raisonnement à toutes les tâches du système, nous pouvons déduire que la synchronisation de toutes les tâches de la VM et de la tâche fictive $\overline{\rho^j}$ nous permet d'obtenir le pire temps de réponse.

La tâche fictive $\overline{\rho^j}$ représentant les inactivités de la VM, nous pouvons conclure que la synchronisation des tâches à la fin de l'exécution de la VM nous permet de définir un candidat d'instant critique. Dans le cas précis d'une VM ordonnancée de manière strictement périodique, il n'existe donc qu'un seul candidat d'instant critique.

Nous pouvons maintenant reprendre les démonstrations précédentes pour les appliquer au cas général.

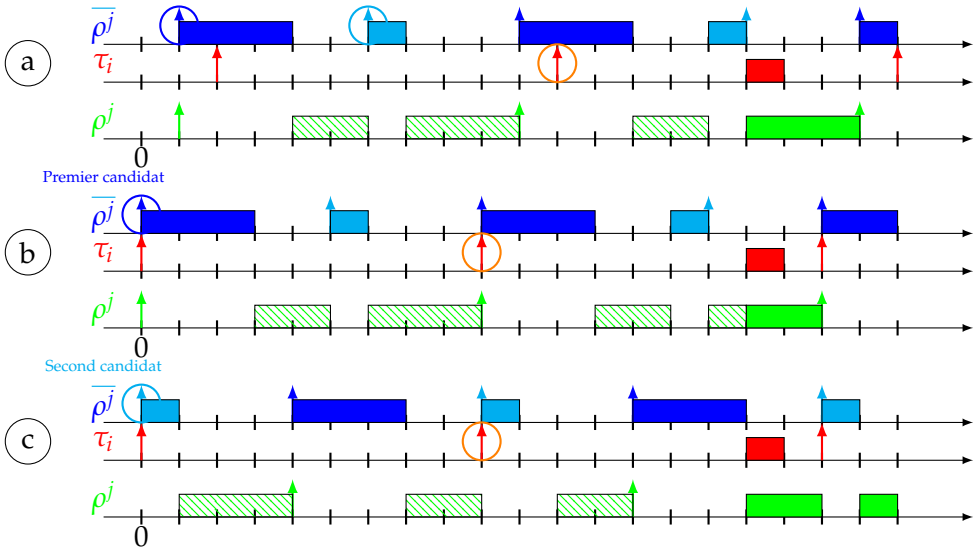


FIGURE 3.8 : Candidats d'instant critique dans une VM ordonnancée avec deux exécutions par période

Preuve lors du cas général : La démonstration des candidats d'instant critique d'une VM dans le cas général utilise encore une fois la tâche fictive $\overline{\rho^j}$. Mais cette fois, la tâche fictive aura plusieurs activations distinctes qui se répètent à chaque période de la VM. Il est donc possible de découper cette tâche en plusieurs sous tâches de priorités égales représentant chaque inactivité de la VM.

Nous donnons un exemple dans le schéma (a) de la Figure 3.8. Nous avons ainsi une VM ρ^j qui a deux exécutions distinctes au cours de sa période. Cela nous donne donc une tâche fictive $\overline{\rho^j}$ exécutées lors de ces deux inactivités. Il est donc possible de représenter cette tâche en deux sous tâches ici représentées en bleu foncé et en cyan. Ces tâches sont ainsi activées au moment de leur exécution. Leur exécution se répétant dans le temps du fait de la période de la VM, il s'agit de tâches périodiques. L'étude est toujours effectuée dans une busy period de niveau i, et les moments où

la VM exécute des tâches de priorité supérieure à τ_i sont toujours représentés en hachuré.

Nous nous intéressons comme précédemment à la seconde instance de la tâche τ_i. Cette fois, il ne sera pas possible de synchroniser cette tâche avec les deux sous tâches de $\overline{\rho^j}$, puisque celles-ci représentent les inactivités de la VM. La synchronisation ne peut être faite qu'avec une des deux sous tâches à la fois. Cela nous donne donc deux possibilités explicitées dans les deux schémas (b) et (c) : soit la tâche τ_i est synchronisée à l'instant 0 avec la sous tâche de $\overline{\rho^j}$ représentée en bleu (premier candidat), soit elle est synchronisée avec la sous tâche représentée en cyan (second candidat).

Lors de ces deux synchronisations, les résultats doivent être comparés avec le scénario non synchrone (représenté par le schéma (a)). Lors de ces deux synchronisations, deux possibilités existent :

- Soit le temps de réponse de l'instance de tâche τ_i ne change pas.
- Soit le temps de réponse de l'instance de tâche τ_i augmente.

Comme précédemment, la synchronisation de la tâche τ_i avec les deux tâches fictives de $\overline{\rho^j}$ nous permettent donc d'obtenir les candidats d'instant critique. Et ainsi de déduire à partir de ceux-ci le temps de réponse pire cas, en étudiant le temps de réponse sur chacun de ces candidats pour déterminer le temps de réponse pire cas.

Encore une fois, ce raisonnement peut être étendu aux autres tâches du système pour déterminer que le pire cas est bien le cas synchrone entre toutes les tâches de la VM étudiée et d'une des sous tâches fictives de $\overline{\rho^j}$.

Il n'est pas possible de plus réduire la liste de candidats à l'instant critique. Pour une VM ρ^j donnée, les nc candidats d'instant critique ρ^j_C sont notés :
$\rho^j_{C_1}, \rho^j_{C_2}, \dots \rho^j_{C_{nc}}$

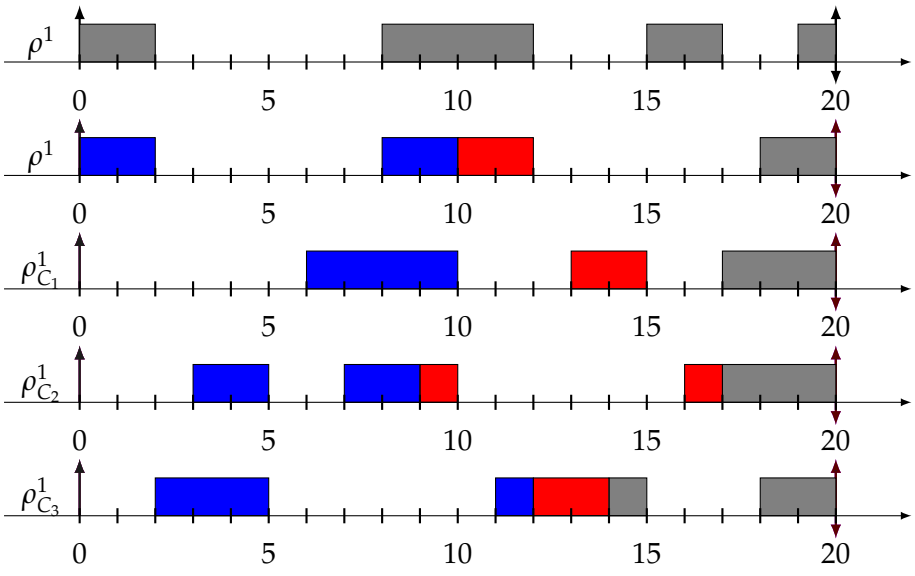


FIGURE 3.9 : Exemples de candidats d'instant critique pour une VM ρ^1 à ordonnancement connu et comme jeu de tâches $\tau_1(4, 20, 20)$ et $\tau_2(2, 20, 20)$

En effet, comme montré dans la Figure 3.9, l'instant critique n'est pas toujours trivial à déterminer. Dans cette figure, nous avons d'abord défini un ordonnancement arbitraire pour la VM ρ^1 (ligne 1). À partir de cet ordonnancement, nous avons

d'abord ordonnancé deux tâches, à savoir $\tau_1(4, 20, 20)$ et $\tau_2(2, 20, 20)$. Les priorités des tâches sont ici données par RM, τ_1 est donc la plus prioritaire, et τ_2 la moins prioritaire. Les paramètres des tâches n'ont que peu d'importance ici, mais ils permettent de montrer le temps de réponse de la tâche τ_2, qui est la tâche de basse priorité ici étudiée.

À la Ligne 2, nous montrons l'ordonnancement des tâches sans prendre en compte les candidats d'instant critique, les tâches sont donc actives dès l'instant 0. Le temps de réponse de τ_2 est de 12 unités de temps. Les trois lignes suivantes montrent les trois candidats d'instant critique, à savoir $\rho^1_{C_1}$, $\rho^1_{C_2}$ et $\rho^1_{C_3}$. Ces candidats sont formés à partir de l'ordonnancement de base, qui sera juste décalé et répété dans le temps. Nous considérons ici que l'instant critique commence toujours à l'instant 0 pour simplifier le schéma. En réalité, par rapport à l'ordonnancement donné en première ligne, les candidats d'instant critique devraient commencer à l'instant 2, 12 et 17. Puisque l'ordonnancement se répète dans le temps, nous pouvons, pour simplifier l'étude, considérer que les candidats d'instant critique commencent à l'instant 0.

Ainsi, nous illustrons dans les trois dernières lignes l'ordonnancement de τ_1 et de τ_2 dans les trois différents cas de candidats d'instant critique. L'ordre des candidats d'instant critique est arbitraire. Cette figure permet de montrer que l'instant critique n'est pas trivial à déterminer : le début de l'exécution de la VM dans le candidat d'instant critique $\rho^1_{C_1}$ est le plus éloigné de l'instant 0, mais le temps de réponse de τ_2 est de 15 unités de temps. En revanche, le temps de réponse de τ_2 avec le candidat d'instant critique $\rho^1_{C_2}$ est le plus long, de 17 unités de temps, alors que le début de l'exécution de la VM est moins éloigné de l'instant 0 que pour le candidat $\rho^1_{C_1}$. Le candidat $\rho^1_{C_3}$, quant à lui, donne un temps de réponse pour τ_2 de 14 unités de temps, soit le temps de réponse le plus court alors que le début de l'exécution de la VM, pour ce candidat, est le plus proche de l'instant 0.

3.3 Temps de réponse pire cas de VMs strictement périodiques

Dans cette section, nous définissons le temps de réponse pire cas de VMs strictement périodiques. Une VM strictement périodique, comme défini dans la Section 3.1 permet de créer une VM particulière : elle n'aura en effet qu'un seul candidat d'instant critique conduisant à une seule exécution non préemptive à chaque période d'activation de la VM. Nous montrons un exemple dans la Figure 3.10. En effet, la VM étant ordonnancée de manière strictement périodique, son ordonnancement se répète entre chaque exécution. Ainsi, nous pouvons visualiser son ordonnancement comme ne contenant qu'une seule exécution qui se répète dans le temps. L'ordonnancement de la VM étant ainsi connu, il est possible de l'analyser sans tenir compte des autres VMs.

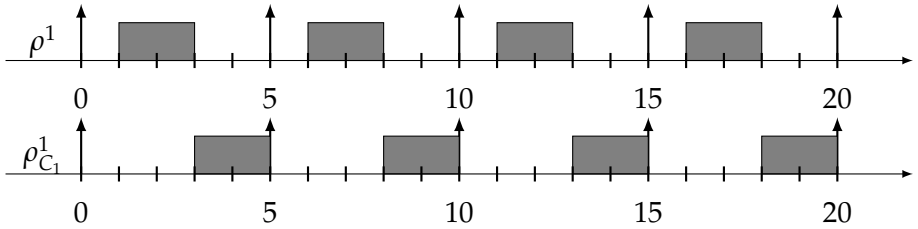


FIGURE 3.10 : Ordonnancement strictement périodique de la VM $\rho^1(2, 5)$ et représentation de son instant critique $\rho^1_{C_1}$

Dans la Figure 3.10, nous avons défini une VM strictement périodique avec les paramètres suivants : $\rho^1(2,5)$. Elle s'exécute donc pour deux unités de temps toutes les cinq unités de temps. Pour information, les schémas d'ordonnancement de candidats d'instant critique sont toujours du même type :

- ils débutent par un temps où la VM n'est pas exécutée ;
- ils se terminent par un temps où la VM est exécutée.

Ainsi, deux VMs strictement périodiques avec les mêmes paramètres auront des instants critiques donnant des ordonnancements identiques.

Nous allons commencer par rappeler la formule de temps de réponse issue de l'état de l'art dans un modèle mono-processeur non hiérarchique. L'analyse de temps de réponse se fait donc à l'instant critique, avec activation synchrone des tâches. Si les tâches sont sporadiques, elles sont alors considérées en pire cas comme des tâches périodiques. Il existe plusieurs formules de temps de réponse, selon l'échéance des tâches : si l'échéance est sur requête, contrainte ou arbitraire. Nous nous intéressons ici aux échéances arbitraires, ce modèle étant le plus général et incluant tous les autres.

Dans l'état de l'art (cf. Chapitre 2.2 [161]), le temps de réponse pire cas r_i d'une tâche τ_i ordonnancée par un ordonnancement à priorités fixes arbitraires au niveau des tâches sur une architecture mono-processeur est défini ainsi :

$$r_i = \max_{\forall q \in \{0,\dots,Q\}} (w_{i,q} - q \times T_i) \tag{3.1}$$

Où q est le numéro de l'instance de la tâche τ_i activée en $q \times T_i$ et Q est le nombre maximum d'instances de tâches à considérer pour trouver le temps de réponse pire cas tel que :

$$w_{i,Q} \leq (Q+1) \times T_i \tag{3.2}$$

Et :

$$w_{i,q}^{m+1} = (q+1) \times C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_{i,q}^m}{T_j} \right\rceil \times C_j \tag{3.3}$$

Où $hp(i)$ est une fonction qui permet de trouver toutes les tâches de priorité supérieure ou égale à la tâche τ_i (sauf τ_i) dans le jeu de tâches τ.

$w_{i,Q}$ correspond à la fin de la première période d'activité du processeur démarrant à l'instant 0 lorsque les tâches sont synchrones en 0. Cette période d'activité est composée d'instances de tâches de priorité fixe supérieure ou égale à τ_i.

La fonction $w_{i,q}^{m+1}$ est une suite croissante et bornée par le plus petit commun multiples des périodes des tâches, elle est donc convergente (lorsque l'utilisation processeur du jeu de tâches est inférieure ou égale à 100%).

La suite est initialisée au rang 0 pour $q = 0$ par $w_{i,0}^0 = C_i$. Lorsque $q > 0$, la suite est initialisée par $w_{i,q}^0 = w_{i,q-1}$. Pour une instance donnée activée en $q \times T_i$, la convergence de la suite est obtenue lorsqu'il existe un entier $p \geq 0$ tel que $w_{i,q}^{p+1} = w_{i,q}^p$. Le pire temps de réponse pour la tâche τ_i est alors obtenu en prenant le maximum des temps de réponse obtenus pour toutes les instances de τ_i activée en $q \times T_i$ avec $q = 0, \dots, Q$ tel que $w_{i,Q}$ corresponde à la fin de la période active de niveau i.

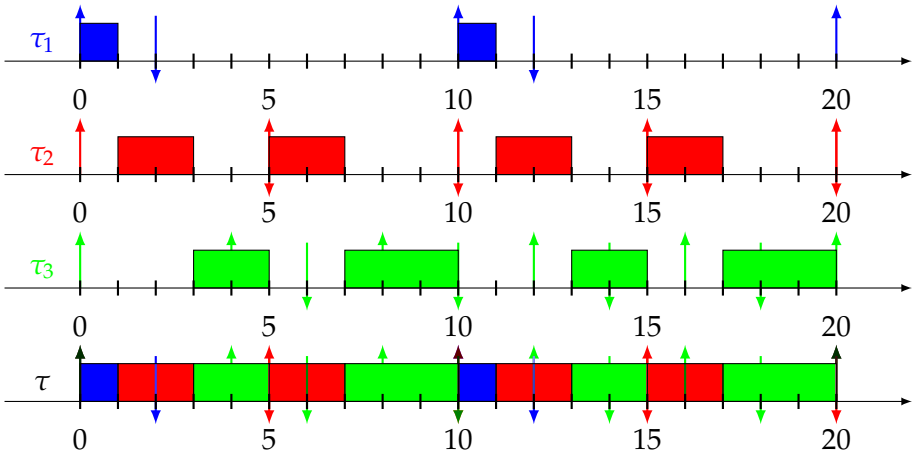


FIGURE 3.11 : Exemple d'ordonnancement du jeu de tâches : $\tau_1(1, 10, 2)$, $\tau_2(2, 5, 5)$ et $\tau_3(2, 4, 6)$ avec comme priorité la plus haute pour τ_1, la plus basse pour τ_3 et une intermédiaire pour τ_2

Nous montrons dans la Figure 3.11 un exemple d'analyse de temps de réponse utilisant les Équations 3.1, 3.2 et 3.3. Nous pouvons d'abord analyser graphiquement cette figure. Le jeu de tâches est ici $\tau_1(1, 10, 2)$, $\tau_2(2, 5, 5)$ et $\tau_3(2, 4, 6)$. La priorité la plus haute est donnée à τ_1, la priorité intermédiaire à τ_2 et la priorité la plus basse à τ_3. Nous obtenons des priorités opposées à RM. Cette assignation de priorités, bien que peu utilisée, nous intéresse pour plusieurs raisons. Premièrement, nous utilisons une assignation de priorité identique dans le Chapitre 6. Deuxièmement, elle permet de voir un jeu de tâches dans lequel le pire temps de réponse n'est pas toujours obtenu pour la première instance .

En effet, les temps de réponse pire cas sont ici de 1 pour τ_1, 3 pour τ_2 et 6 pour τ_3. Nous allons maintenant les calculer grâce aux formules précédentes. Commençons par la tâche la plus prioritaire, à savoir $\tau_1(1, 10, 2)$.

D'après l'Équation 3.3,

$$\begin{aligned} w^0_{1,0} &= 1 \\ w^1_{1,0} &= (0+1) \times 1 \\ &= 1 \end{aligned}$$

Les résultats de $w^0_{1,0}$ étant égaux à ceux de $w^1_{1,0}$, nous avons trouvé le résultat de la suite de récurrence. Puisque la tâche τ_1 est la tâche de plus haute priorité, la somme de l'équation n'a pas lieu d'être. En effet, celle-ci dépend uniquement des tâches de plus haute priorité. De plus, d'après l'Équation 3.2,

$$\begin{aligned} w_{1,0} &\leq (0+1) \times 10 \\ 1 &\leq 10 \end{aligned}$$

Cette inéquation étant vraie, $Q = 0$. Nous pouvons ainsi en déduire le temps de réponse pire cas de l'Équation 3.1 :

$$\begin{aligned} r_1 &= \max(w_{1,0} - 0 \times 10) \\ &= \max(1 - 0 \times 10) \\ &= \max(1) \\ &= 1 \end{aligned}$$

Concernant τ_2 :

$$\begin{aligned} w^0_{2,0} &= 1 \\ w^1_{2,0} &= (0+1) \times 2 + \left\lceil \frac{1}{10} \right\rceil \times 1 \\ &= 2 + 1 \\ &= 3 \\ w^2_{2,0} &= (0+1) \times 2 + \left\lceil \frac{3}{10} \right\rceil \times 1 \\ &= 2 + 1 \\ &= 3 \end{aligned}$$

Les résultats de $w^1_{2,0}$ et $w^2_{2,0}$ étant les mêmes, le résultat est donc de 3. Puisque :

$$\begin{aligned} w_{2,0} &\leq (0+1) \times 5 \\ 3 &\leq 5 \end{aligned}$$

Est vraie, alors nous pouvons nous arrêter sur ce résultat. Ainsi :

$$\begin{aligned} r_2 &= \max(w_{2,0} - 0 \times 5) \\ &= \max(3 - 0 \times 5) \\ &= \max(3) \\ &= 3 \end{aligned}$$

Procédons au calcul de τ_3. Cette fois, comme nous le voyons dans la Figure 3.11, le temps de réponse pire cas est de 6, mais celui-ci n'est pas sur la première instance de tâche de τ_3. Procédons aux calculs :

$$\begin{aligned} w^0_{3,0} &= 1 \\ w^1_{3,0} &= (0+1) \times 2 + \left\lceil \frac{1}{10} \right\rceil \times 1 + \left\lceil \frac{1}{5} \right\rceil \times 2 \\ &= 2 + 1 + 2 \\ &= 5 \\ w^2_{3,0} &= (0+1) \times 2 + \left\lceil \frac{5}{10} \right\rceil \times 1 + \left\lceil \frac{5}{5} \right\rceil \times 2 \\ &= 2 + 1 + 2 \\ &= 5 \end{aligned}$$

Cette fois-ci la condition d'arrêt à vérifier est :

$$w_{3,0} \leq (0+1) \times 4$$
$$5 \leq 4$$

Cette inéquation étant fausse, nous devons continuer le calcul et ainsi incrémenter q de 1.

$$\begin{aligned} w_{3,1}^0 &= 6 \\ w_{3,1}^1 &= (1+1) \times 2 + \left\lceil \frac{6}{10} \right\rceil \times 1 + \left\lceil \frac{6}{5} \right\rceil \times 2 \\ &= 4 + 1 + 4 \\ &= 9 \\ w_{3,1}^2 &= (1+1) \times 2 + \left\lceil \frac{9}{10} \right\rceil \times 1 + \left\lceil \frac{9}{5} \right\rceil \times 2 \\ &= 4 + 1 + 4 \\ &= 9 \end{aligned}$$

Puisque $w_{3,1}^1 = w_{3,1}^2$, la suite a convergé. Vérifions maintenant l'inéquation :

$$w_{3,1} \leq (1+1) \times 4$$
$$9 \leq 8$$

L'inéquation étant encore fausse, nous devons continuer d'incrémenter q de 1.

$$\begin{aligned} w_{3,2}^0 &= 10 \\ w_{3,2}^1 &= (2+1) \times 2 + \left\lceil \frac{10}{10} \right\rceil \times 1 + \left\lceil \frac{10}{5} \right\rceil \times 2 \\ &= 6 + 1 + 4 \\ &= 11 \\ w_{3,2}^2 &= (2+1) \times 2 + \left\lceil \frac{11}{10} \right\rceil \times 1 + \left\lceil \frac{11}{5} \right\rceil \times 2 \\ &= 6 + 2 + 6 \\ &= 14 \\ w_{3,2}^3 &= (2+1) \times 2 + \left\lceil \frac{14}{10} \right\rceil \times 1 + \left\lceil \frac{14}{5} \right\rceil \times 2 \\ &= 6 + 2 + 6 \\ &= 14 \end{aligned}$$

Comme $w_{3,2}^2 = w_{3,2}^3$ le résultat de la suite de récurrence est 14. Testons l'inéquation :

$$w_{3,2} \leq (2+1) \times 4$$
$$14 \leq 12$$

L'inéquation étant encore fausse, nous devons encore incrémenter q de 1. Donc $q = 3$.

$$
\begin{aligned}
w^0_{3,3} &= 14 \\
w^1_{3,3} &= (3+1) \times 2 + \left\lceil \frac{14}{10} \right\rceil \times 1 + \left\lceil \frac{14}{5} \right\rceil \times 2 \\
&= 8 + 2 + 6 \\
&= 16 \\
w^2_{3,3} &= (3+1) \times 2 + \left\lceil \frac{16}{10} \right\rceil \times 1 + \left\lceil \frac{16}{5} \right\rceil \times 2 \\
&= 8 + 2 + 8 \\
&= 18 \\
w^3_{3,3} &= (3+1) \times 2 + \left\lceil \frac{18}{10} \right\rceil \times 1 + \left\lceil \frac{18}{5} \right\rceil \times 2 \\
&= 8 + 2 + 8 \\
&= 18
\end{aligned}
$$

Comme $w^2_{3,3} = w^3_{3,3}$, la suite de récurrence peut s'arrêter. Testons l'inéquation :

$$
\begin{aligned}
w_{3,3} &\leq (3+1) \times 4 \\
18 &\leq 16
\end{aligned}
$$

Celle-ci étant encore fausse, nous devons incrémenter une dernière fois q. Donc $q = 4$.

$$
\begin{aligned}
w^0_{3,4} &= 19 \\
w^1_{3,4} &= (4+1) \times 2 + \left\lceil \frac{19}{10} \right\rceil \times 1 + \left\lceil \frac{19}{5} \right\rceil \times 2 \\
&= 10 + 2 + 8 \\
&= 20 \\
w^2_{3,4} &= (4+1) \times 2 + \left\lceil \frac{20}{10} \right\rceil \times 1 + \left\lceil \frac{20}{5} \right\rceil \times 2 \\
&= 10 + 2 + 8 \\
&= 20
\end{aligned}
$$

Comme $w^1_{3,4} = w^2_{3,4}$ le résultat de la suite de récurrence est trouvé. Vérifions l'inéquation :

$$
\begin{aligned}
w_{3,4} &\leq (4+1) \times 4 \\
20 &\leq 20
\end{aligned}
$$

Celle-ci étant vraie, le calcul des $w_{i,Q}$ peut s'arrêter. Calculons maintenant le temps de réponse d'après l'Équation 3.1 :

$$
\begin{aligned}
r_3 &= \max_{\forall q \in \{0...4\}} (w_{3,q} - q \times 4) \\
&= \max(5 - 0 \times 4; 9 - 1 \times 4; 14 - 2 \times 4; 18 - 3 \times 4; 20 - 4 \times 4) \\
&= \max(5, 5, 6, 6, 4) \\
r_3 &= 6
\end{aligned}
$$

Ainsi, après calcul et vérification graphique sur la Figure 3.11, le temps de réponse pire cas de τ_3 est de 6, et est obtenu pour les instances de tâches activées en qT_i pour $q = 2$ et $q = 3$.

Après avoir rappelé les équations de temps de réponse pire cas dans le cas classique, nous allons intégrer ces travaux dans notre modèle hiérarchique à deux niveaux lorsque la VM n'a qu'un seul instant critique. Pour rappel, le premier niveau, celui des VMs, a un ordonnancement arbitraire, et les tâches exécutées dans chaque VM sont sporadiques avec des échéances arbitraires. Nous étudions l'ordonnnancement des tâches, et non des VMs à ce niveau.

Pour commencer, nous rappelons que les candidats d'instant critique pour un tel modèle sont donnés dans la Définition 42. Dans ce contexte, cela ne correspond qu'à un seul candidat d'instant critique. Nous donnons maintenant l'équation de temps de réponse pire cas pour une tâche τ_i dans la VM ρ^s :

$$r_i = \max_{\forall q \in \{0,\dots,Q\}} (w_{i,q} - q \times T_i) \tag{3.4}$$

De même, q est le numéro de l'instance de tâche activée en $q \times T_i$, et Q est le nombre maximum d'instances de tâches à considérer tel que :

$$w_{i,Q} \leq (Q + 1) \times T_i \tag{3.5}$$

Et :

$$w_{i,q}^{m+1} = (q + 1) \times C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_{i,q}^m}{T_j} \right\rceil \times C_j + \left\lceil \frac{w_{i,q}^m}{T^s} \right\rceil (T^s - C^s) \tag{3.6}$$

Le changement dans l'équation itérative se situe dans la dernière partie de l'Équation 3.6. En effet, dans cette équation un élément en plus est ajouté. Pour simplifier, il peut être vu comme le temps où la VM étudiée n'est pas ordonnancée : $T^s - C^s$, correspondant au temps pendant lequel la VM n'est pas exécutée, T^s étant sa période et C^s la durée du slot où la VM est exécutée. Nous ajoutons ainsi au temps de réponse le temps pendant lequel la tâche n'a pas pu être exécutée, par rapport à son temps de réponse actuel, contenu dans $w_{i,q}^m$.

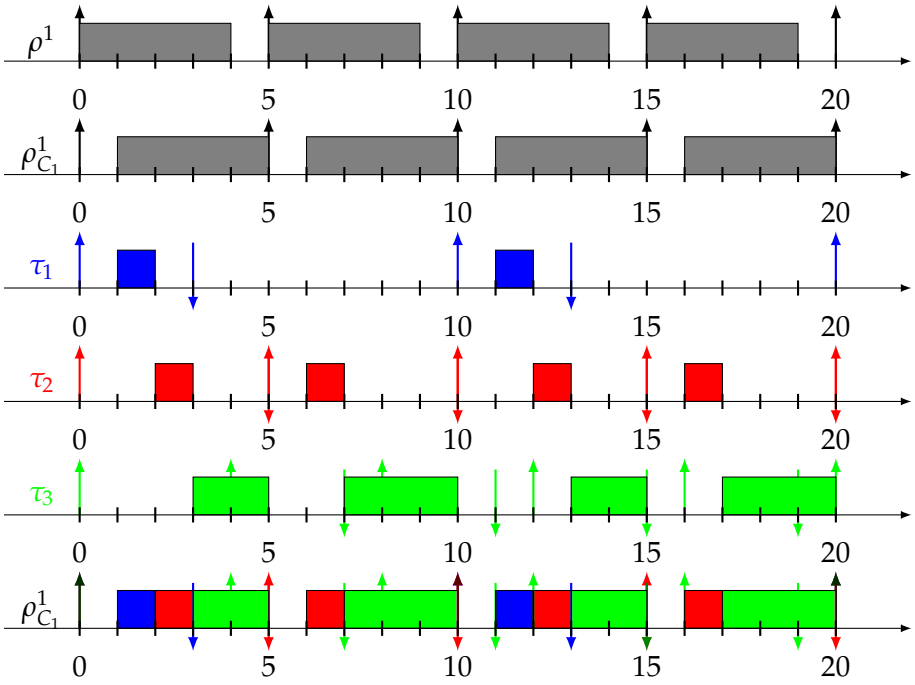


FIGURE 3.12 : Temps de réponse pire cas dans un modèle hiérarchique avec comme VM $\rho^1(4,5)$ et comme jeu de tâches classé par ordre de priorités $\tau_1(1,10,3)$, $\tau_2(1,5,5)$ et $\tau_3(2,4,7)$

Nous montrons dans la Figure 3.12 un exemple de temps de réponse dans une VM $\rho^1(4,5)$ (Ligne 1) avec un seul candidat d'instant critique $\rho^1_{C_1}$ (Ligne 2). Cette VM contient le jeu de tâches suivant, classé de la tâche la plus prioritaire à la moins prioritaire : $\tau_1(1,10,3)$, $\tau_2(1,5,5)$ et $\tau_3(2,4,7)$. Nous pouvons grâce aux Lignes 3, 4 et 5 de la Figure 3.12 prévisualiser les temps de réponse pire cas des différentes tâches. À savoir, 2 pour τ_1, 3 pour τ_2 et 6 pour τ_3. Nous ne donnerons en exemple ici que les calculs du temps de réponse pire cas de τ_3, celle-ci étant la plus intéressante. En effet, cette tâche est la tâche à la plus basse priorité et donc la plus préemptée.

Commençons par reprendre l'Équation 3.6 :

$$
\begin{aligned}
w^0_{3,0} &= 1 \\
w^1_{3,0} &= (0+1)\times 2 + \left\lceil \frac{1}{10} \right\rceil \times 1 + \left\lceil \frac{1}{5} \right\rceil \times 1 + \left\lceil \frac{1}{5} \right\rceil \times (5-4) \\
&= 5 \\
w^2_{3,0} &= (0+1)\times 2 + \left\lceil \frac{5}{10} \right\rceil \times 1 + \left\lceil \frac{5}{5} \right\rceil \times 1 + \left\lceil \frac{5}{5} \right\rceil \times (5-4) \\
&= 5
\end{aligned}
$$

$w^1_{3,0}$ étant égal à $w^2_{3,0}$, la suite a convergé. D'après l'Équation 3.5, nous testons si nous devons continuer ou non :

$$
\begin{aligned}
5 &\leq (0+1)\times 4 \\
5 &\leq 4
\end{aligned}
$$

q	Équation 3.6	Équation 3.5
0	5	$5 \leq 4$
1	9	$9 \leq 8$
2	14	$14 \leq 12$
3	18	$18 \leq 16$
4	20	$20 \leq 20$

TABLE 3.1 : Temps de réponses de la tâche τ_3

Cette inéquation étant fausse, nous devons continuer le calcul en incrémentant le q de 1. Pour simplifier, nous passons directement à la dernière étape : $q = 4$.

$$\begin{aligned} w_{3,4}^0 &= 18 + 1 \\ &= 19 \\ w_{3,4}^1 &= (4+1) \times 2 + \left\lceil \frac{19}{10} \right\rceil \times 1 + \left\lceil \frac{19}{5} \right\rceil \times 1 + \left\lceil \frac{19}{5} \right\rceil \times 1 \\ &= 20 \\ w_{3,4}^2 &= (4+1) \times 2 + \left\lceil \frac{20}{10} \right\rceil \times 1 + \left\lceil \frac{20}{5} \right\rceil \times 1 + \left\lceil \frac{20}{5} \right\rceil \times 1 \\ &= 20 \end{aligned}$$

Comme $w_{3,4}^1 = w_{3,4}^2$, la suite de récurrence est terminée. Vérifions l'Équation 3.5 :

$$\begin{aligned} 20 &\leq (4+1) \times 4 \\ 20 &\leq 20 \end{aligned}$$

Celle-ci étant vraie, les étapes sont ici terminées.

Nous résumons dans la Table 3.1 les différentes étapes pour trouver le temps de réponse de τ_3 défini par l'Équation 3.4 :

$$\begin{aligned} r_3 &= \max_{\forall q \in \{0,\dots,4\}} (5 - 0 \times 4; 9 - 1 \times 4; 14 - 2 \times 4; 18 - 3 \times 4; 20 - 4 \times 4) \\ &= 6 \end{aligned}$$

Le temps de réponse de τ_3 donné par l'Équation 3.1 est donc bien le même que celui montré dans la Figure 3.12, à savoir de 6 unités de temps. Il se fait sur les instances de tâches $q = 2$ et $q = 3$. Cela met ainsi en avant, qu'y compris dans ce modèle, le temps de réponse pire cas n'est pas nécessairement la première instance de tâche lorsque les tâches ont des échéances arbitraires.

Si nous ajoutons les overheads (Définition 15) à la VM $\rho^s(C^s, T^s, O^s)$ étudiée, le temps de réponse de la tâche τ_1 devient alors :

$$r_i = \max_{\forall q \in \{0,\dots,Q\}} (w_{i,q} - q \times T_i) \tag{3.7}$$

Où q et Q ne changent pas, et

$$w_{i,q} \leq (Q+1) \times T_i$$

Et :

$$w_{i,q}^{m+1} = (q+1) \times C_i + \sum_{j \in hp(i)} \left\lceil \frac{w_{i,q}^{m}}{T_j} \right\rceil \times C_j + \left\lceil \frac{w_{i,q}^{m}}{T^s} \right\rceil \times (T^s - C^s + O^s)$$

En sachant qu'avec les overheads, le temps d'exécution C^s d'une VM inclus le temps d'exécution C^s d'une VM contenant le temps ajouté par l'overhead O^s, nous montrons dans la Figure 3.13 un exemple de VM ordonnancée avec overhead. Il s'agit de la VM $\rho^1(3,5,1)$: son temps d'exécution est donc de 3, sa période de 5 et son overhead de 1. Comme montré dans le schéma, l'overhead est inclus dans dans C^s. En effet, son temps d'exécution est de 3 unités de temps, son overhead de 1. Le temps d'exécution effectif pendant lequel les tâches peuvent être ordonnancées est donc de $3 - 1 = 2$ unités de temps.

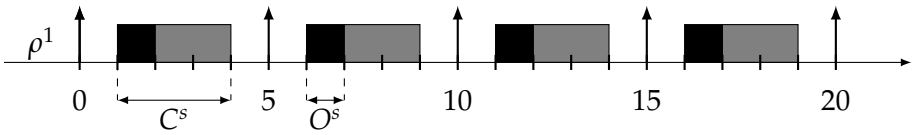


FIGURE 3.13 : VM $\rho^1(3,5,1)$ avec un overhead de 1

Des exemples de calculs de temps de réponse pour cette nouvelle formule (Équation 3.3) ne seront pas donnés, celle-ci étant similaire au temps de réponse pire cas sans overhead. En effet, le seul changement apporté ici est l'inclusion des overheads, qui s'ajoute directement au moment d'inclure le temps où la VM étudiée n'est pas exécutée, à savoir $T^S - C^s$, qui devient alors $T^s - C^s + O^s$.

3.4 Temps de réponse pire cas dans le cas général

Nous introduisons dans cette partie une formule de temps de réponse pire cas pour notre modèle. Cette fois, cette formule n'aura aucune contrainte d'utilisation : elle permettra d'obtenir le temps de réponse pire cas dans le cas d'un nombre arbitraire d'intervalles activés périodiquement, conduisant à un nombre arbitraire de candidats d'instant critique pour la VM.

La première étape consiste à déterminer les scénarios de candidats d'instant critique. Pour cela, nous partons d'un ordonnancement quelconque. À partir de cet ordonnancement quelconque, nous devons ensuite déterminer les scénarios de candidats d'instant critique. Comme précédemment défini dans la Définition 42, les candidats d'instant critique débutent à chaque fin d'exécution de la VM.

Définition 43 (Scénario de candidat d'instant critique)
Un scénario de candidat d'instant critique est un scénario d'activation particulier qui permet de définir le temps de réponse pire cas calculé par l'activation des tâches à l'instant critique. Le scénario est ainsi retravaillé pour débuter à l'instant 0. Il suffit d'effectuer un décalage temporel du scénario racine.

Définition 44 (Scénario racine)
Le scénario racine est le scénario de départ qui permet de déterminer les scénarios de candidats d'instant critique. Ce scénario est donc celui trouvé par l'ordonnancement des VMs avant d'être retravaillé pour être analysé.

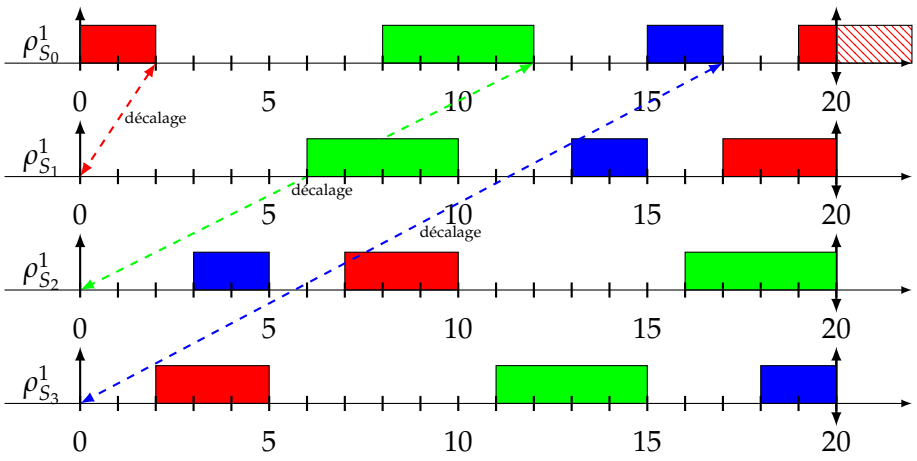


FIGURE 3.14 : Exécution de la VM $\rho^1_{S_0}$ et explicitation de ses scénarios de candidats d'instant critique $\rho^1_{S_1}$, $\rho^1_{S_2}$ et $\rho^1_{S_3}$ formés à partir du scénario $\rho^1_{S_0}$

Nous montrons dans la Figure 3.14 comment définir les scénarios des candidats de la VM ρ^1. Il s'agit d'un décalage temporel prenant en compte le scénario racine noté $\rho^1_{S_0}$. Ce scénario est montré dans la Figure 3.14 à la première ligne. Le premier scénario, $\rho^1_{S_1}$ correspond ainsi au scénario de la VM ρ^1_S mais avec un décalage temporel de 2 unités de temps. Cela permet ainsi de débuter le scénario de candidat d'instant critique sur une fin d'exécution de la VM et de le terminer sur une exécution de la VM. C'est sous cette forme que doivent se trouver les scénarios de candidats d'instant critique. Cela nous donne donc pour $\rho^1_{S_2}$ un décalage temporel de 12 unités de temps et de 17 unités de temps pour $\rho^1_{S_3}$: tous les débuts de scénarios de candidats d'instant critique se font à un décalage temporel correspondant au début d'une fin d'exécution de la VM. Bien qu'il y ait quatre exécutions visibles sur le scénario racine $\rho^1_{S_0}$, il n'en existe que trois. En effet, puisque les exécutions de la VM se répètent dans le temps et que le scénario commence et se termine par une exécution de la VM, un décalage temporel comme fait lors de tous les scénarios de candidat d'instant critique met en évidence qu'il n'y a que trois exécutions distinctes.

Pour commencer, nous avons besoin de plus de notations pour définir les VMs avec plusieurs scénarios de candidats d'instant critique définis dans l'ensemble ρ^i_S. En effet, nous notons $\Psi^i_{S_n}$ l'ensemble des différents intervalles d'exécution de la VM $\rho^i(C^i, T^i)$ pour un scénario de candidat d'instant critique donné $\rho^i_{S_n}$. Le $k^{ème}$ élément de $\Psi^i_{S_n}$ se définit ainsi :

$$\psi_k(S_k, I_k)$$

Où S_k correspond à la date de début ("start") de l'intervalle, et I_k correspond à la durée d'inactivité de la VM au début de l'intervalle d'exécution.

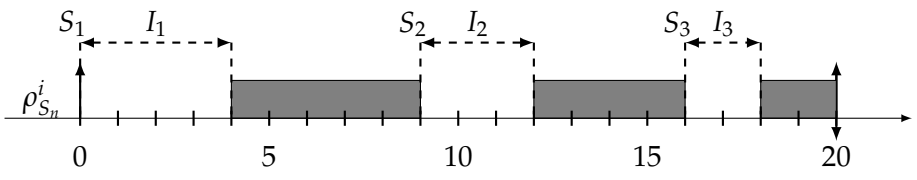


FIGURE 3.15 : VM ρ^i avec plusieurs candidats d'instant critique et définitions des éléments des intervalles d'exécution

Nous présentons dans la Figure 3.15 un exemple pour une VM ρ^i avec trois candidats d'instant critique. Le scénario de candidat d'instant critique de la VM ρ^i se répète au bout de 20 unités de temps, soit T^i. Ainsi, la VM n'a que 3 intervalles d'exécution : $\psi_1(0,4)$, $\psi_2(9,3)$ et $\psi_3(16,2)$. Les intervalles des exécutions ne peuvent se définir que sur des scénarios de candidats d'instant critique : ils débutent sur un temps où la VM n'est pas exécutée et se terminent sur une exécution.

Le temps de réponse pire cas d'une tâche τ_i dans la VM ρ^s avec comme ensemble de scénarios de candidats d'instant critique ρ^s_S dans ce modèle devient alors :

$$r_i = \max_{\forall \rho^s_{S_k} \in \rho^s_S} (r_{i,k}) \tag{3.8}$$

Où $r_{i,k}$ correspond au temps de réponse pire cas de la tâche τ_i pour le candidat d'instant critique $\rho^s_{S_k}$ et où :

$$r_{i,k} = \max_{\forall q \in \{0,\dots,Q\}} (w_{i,q,k} - q \times T_i) \tag{3.9}$$

Ici $w_{i,q,k}$ est le temps de réponse de la tâche τ_i, q correspond toujours au numéro de l'instance de tâche observée et k est le $k^{ème}$ scénario de candidat d'instant critique de la VM. Q se définit tel que :

$$w_{i,Q,k} \leq (Q+1) \times T_i \tag{3.10}$$

Et :

$$w^{m+1}_{i,q,k} = (q+1) \times C_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{w^m_{i,q,k}}{T_j} \right\rceil \times C_j \right) + \sum_{\psi_l \in \Psi^s_{S_k}} \left(\left\lceil \frac{w^m_{i,q,k} - S_l}{T^s} \right\rceil \times I_l \right) \tag{3.11}$$

Par rapport au temps de réponse sans modèle hiérarchique, la partie qui change ici est la seconde somme, à savoir, $\sum_{\psi_l \in \Psi^s_{S_s}} \left(\left\lceil \frac{w^m_{i,q} - S_l}{T^s} \right\rceil \times I_l \right)$. Pour résumer cette partie, il s'agit, de la somme des moments où la VM n'est pas exécutée, dans tous les intervalles d'exécution. Le terme $w^m_{i,q,k}$ correspond ici au résultat précédent de la suite lors du calcul , S_l correspond donc au début de l'intervalle d'exécution, qui peut être vu comme un offset, T_s correspond à la période de répétition du schéma de la VM et I_l au temps d'inactivité de l'intervalle d'exécution observé.

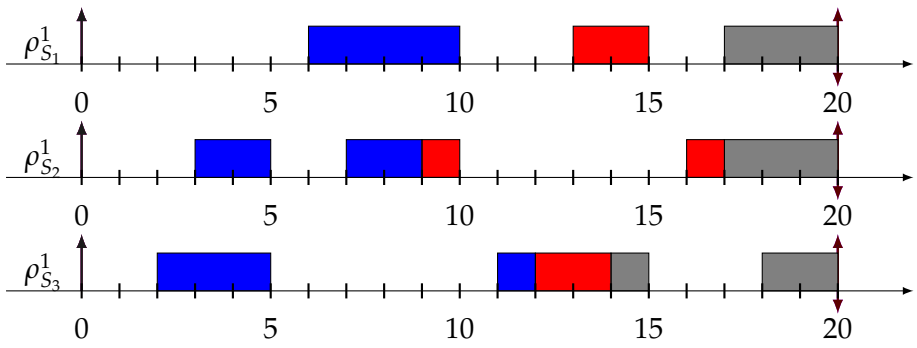


FIGURE 3.16 : Temps de réponses sur la VM $\rho^1(9,20)$ avec comme jeu de tâches $\tau_1(4,20,20)$ et $\tau_2(2,20,20)$

$\Psi^1_{S_1}$:	$\psi_1(0,6)$,	$\psi_2(10,3)$ et	$\psi_3(15,2)$
$\Psi^1_{S_2}$:	$\psi_1(0,3)$,	$\psi_2(5,2)$ et	$\psi_3(10,6)$
$\Psi^1_{S_3}$:	$\psi_1(0,2)$,	$\psi_2(5,6)$ et	$\psi_3(15,3)$

TABLE 3.2 : Intervalles d'exécution de la VM $\rho^1(9,20)$ de la Figure 3.16

Dans la Figure 3.16, nous présentons un ordonnancement donné pour la VM $\rho^1(9,20)$. Cet ordonnancement donne trois candidats d'instant critique, donc trois scénarios montrés sur les trois différentes lignes. Nous ordonnançons ensuite les tâches $\tau_1(4,20,20)$ et $\tau_2(2,20,20)$ au sein des scénarios de candidats d'instant critique. Ici, la première instance de tâche suffira à trouver les temps de réponse.

Les temps de réponse étant montrés dans la Figure 3.16, nous cherchons juste à vérifier le calcul. Nous ne calculerons que le temps de réponse de τ_2, cette tâche étant la plus préemptée, sa priorité étant plus basse.

Nous décrivons dans la Table 3.2 l'ensemble des intervalles d'exécution des scénarios de candidats d'instant critique Ψ^1_S. Après avoir défini ces trois ensembles, il est possible de procéder aux calculs de temps de réponse pour le scénario de candidat d'instant critique $\rho^1_{S_1}$. Reprenons l'Équation 3.11 pour la tâche τ_2 :

$$
\begin{aligned}
w^0_{2,0,1} &= 0 \\
w^1_{2,0,1} &= (0+1)\times 2 + \left\lceil \frac{1}{20} \right\rceil \times 4 + \left\lceil \frac{1-0}{20} \right\rceil \times 6 + \left\lceil \frac{1-10}{20} \right\rceil \times 3 + \left\lceil \frac{1-15}{20} \right\rceil \times 2 \\
&= 2+4+6+0+0 \\
&= 12 \\
w^2_{2,0,1} &= (0+1)\times 2 + \left\lceil \frac{12}{20} \right\rceil \times 4 + \left\lceil \frac{12-0}{20} \right\rceil \times 6 + \left\lceil \frac{12-10}{20} \right\rceil \times 3 + \left\lceil \frac{12-15}{20} \right\rceil \times 2 \\
&= 2+4+6+3+0 \\
&= 15 \\
w^3_{2,0,1} &= (0+1)\times 2 + \left\lceil \frac{15}{20} \right\rceil \times 4 + \left\lceil \frac{15-0}{20} \right\rceil \times 6 + \left\lceil \frac{15-10}{20} \right\rceil \times 3 + \left\lceil \frac{15-15}{20} \right\rceil \times 2 \\
&= 2+4+6+3+0 \\
&= 15
\end{aligned}
$$

Puisque $w^2_{2,0,1} = w^3_{2,0,1}$, nous vérifions ensuite l'Équation 3.10 :

$$
\begin{aligned}
w_{2,0,1} &\leq (0+1)\times 20 \\
15 &\leq 20
\end{aligned}
$$

Cette inéquation étant vraie, nous pouvons passer à l'Équation 3.9 :

$$
\begin{aligned}
r_{2,1} &= \max_{\forall q \in \{0\}} (15 - 0 \times 20) \\
&= 15
\end{aligned}
$$

Procédons maintenant aux calculs de temps de réponse pire cas pour les autres candidats d'instant critique. Pour $\rho^1_{S_2}$:

$$
\begin{aligned}
w^0_{2,0,2} &= 1 \\
w^1_{2,0,2} &= (0+1)\times 2 + \left\lceil \frac{1}{20} \right\rceil \times 4 + \left\lceil \frac{1-0}{20} \right\rceil \times 3 + \left\lceil \frac{1-5}{20} \right\rceil \times 2 + \left\lceil \frac{1-10}{20} \right\rceil \times 6 \\
&= 2+4+3+0+0 \\
&= 9 \\
w^2_{2,0,2} &= (0+1)\times 2 + \left\lceil \frac{9}{20} \right\rceil \times 4 + \left\lceil \frac{9-0}{20} \right\rceil \times 3 + \left\lceil \frac{9-5}{20} \right\rceil \times 2 + \left\lceil \frac{9-10}{20} \right\rceil \times 6 \\
&= 2+4+3+2+0 \\
&= 11 \\
w^3_{2,0,2} &= (0+1)\times 2 + \left\lceil \frac{11}{20} \right\rceil \times 4 + \left\lceil \frac{11-0}{20} \right\rceil \times 3 + \left\lceil \frac{11-5}{20} \right\rceil \times 2 + \left\lceil \frac{11-10}{20} \right\rceil \times 6 \\
&= 2+4+3+2+6 \\
&= 17 \\
w^4_{2,0,2} &= (0+1)\times 2 + \left\lceil \frac{17}{20} \right\rceil \times 4 + \left\lceil \frac{17-0}{20} \right\rceil \times 3 + \left\lceil \frac{17-5}{20} \right\rceil \times 2 + \left\lceil \frac{17-10}{20} \right\rceil \times 6 \\
&= 2+4+3+2+6 \\
&= 17
\end{aligned}
$$

Cette fois, $w^3_{2,0,2} = w^4_{2,0,2}$, le résultat est donc trouvé. Testons l'Équation 3.10 :

$$
\begin{aligned}
w_{2,0,2} &\leq (0+1)\times 20 \\
17 &\leq 20
\end{aligned}
$$

Cette inéquation étant également vraie, nous pouvons alors trouver le temps de réponse dans le cas du scénario candidat d'instant critique $\rho^1_{S_2}$ via l'Équation 3.9 :

$$
\begin{aligned}
r_{2,2} &= \max_{\forall q \in \{0\}} (17 - 0 \times 20) \\
&= 17
\end{aligned}
$$

Il ne nous reste maintenant plus qu'à calculer le temps de réponse pire cas pour le troisième scénario de candidat d'instant critique $\rho^1_{S_3}$. Reprenons l'Équation 3.11 :

$$
\begin{aligned}
w^0_{2,0,3} &= 1 \\
w^1_{2,0,3} &= (0+1)\times 2 + \left\lceil \frac{1}{20} \right\rceil \times 4 + \left\lceil \frac{1-0}{20} \right\rceil \times 2 + \left\lceil \frac{1-5}{20} \right\rceil \times 6 + \left\lceil \frac{1-15}{20} \right\rceil \times 3 \\
&= 2+4+2+0+0 \\
&= 8 \\
w^2_{2,0,3} &= (0+1)\times 2 + \left\lceil \frac{8}{20} \right\rceil \times 4 + \left\lceil \frac{8-0}{20} \right\rceil \times 2 + \left\lceil \frac{8-5}{20} \right\rceil \times 6 + \left\lceil \frac{8-15}{20} \right\rceil \times 3 \\
&= 2+4+2+6+0 \\
&= 14 \\
w^3_{2,0,3} &= (0+1)\times 2 + \left\lceil \frac{14}{20} \right\rceil \times 4 + \left\lceil \frac{14-0}{20} \right\rceil \times 2 + \left\lceil \frac{14-5}{20} \right\rceil \times 6 + \left\lceil \frac{14-15}{20} \right\rceil \times 3 \\
&= 2+4+2+6+0 \\
&= 14
\end{aligned}
$$

Comme $w^2_{2,0,3} = w^3_{2,0,3}$, nous pouvons maintenant vérifier l'Équation 3.10 :

$$
\begin{aligned}
w_{2,0,3} &\leq (0+1)\times 20 \\
14 &\leq 20
\end{aligned}
$$

Cette inéquation étant vraie, nous obtenons le temps de réponse du troisième candidat d'instant critique grâce à l'Équation 3.9 :

$$
\begin{aligned}
r_{2,3} &= \max_{\forall q \in \{0\}} (14 - 0 \times 20) \\
&= 14
\end{aligned}
$$

Nous pouvons maintenant obtenir le temps de réponse pire cas de la VM ρ^1 grâce à l'Équation 3.8 :

$$
\begin{aligned}
r_2 &= \max_{\forall \rho^1_{S_k} \in \rho^1_S} (r_{2,k}) \\
&= \max(r_{2,1}, r_{2,2}, r_{2,3}) \\
&= \max(15, 17, 14) \\
&= 17
\end{aligned}
$$

Pour l'intégration des overheads des VMs $\rho^s(C^s, T^s, O^s)$, comme précédemment, le seul changement se fait sur l'Équation 3.11 qui devient alors :

$$
w^{m+1}_{i,q,k} = (q+1)\times C_i + \sum_{j \in hp(i)} \left(\left\lceil \frac{w^m_{i,q,k}}{T_j} \right\rceil \times C_j \right) + \sum_{\psi_l \in \Psi^s_{S_k}} \left(\left\lceil \frac{w^m_{i,q,k} - S_l}{T^s} \right\rceil \times (I_l + O^s) \right) \tag{3.12}
$$

Ainsi, grâce à ces formules de temps de réponse, nous pouvons définir pour notre modèle hiérarchique à deux niveaux tous les temps de réponse pour les tâches

à l'intérieur des VMs si leur ordonnancement est donné. Ces formules de temps de réponse nous serviront tout au long de la thèse pour déterminer les ordonnancements et les paramètres des VMs. En effet, puisque nous pourrons alors vérifier si les tâches respectent leurs échéances, nous déterminerons alors si les paramètres des VMs sont utilisables ou non, et ainsi choisir d'autres paramètres ou d'autres façons d'ordonnancer les VMs.

3.5 Conditions d'ordonnançabilité

Nous verrons dans cette section différentes conditions d'ordonnançabilité pour les VMs. Ces conditions d'ordonnançabilité seront trouvées, pour certaines, à partir des formules de pire temps de réponse des tâches définies dans les Sections 3.3 et 3.4.

Ces conditions nous serviront à définir les tables d'ordonnancements des VMs. Une première condition nécessaire simple consiste à vérifier que l'utilisation totale des VMs est inférieure à 1 :

$$\sum_{\forall \rho^s \in \rho} U^s \leq 1 \tag{3.13}$$

Où ρ est l'ensemble des VMs du système et U^s correspond à l'utilisation totale des tâches de la VM ρ^s. Ainsi,

$$U^s = \sum_{\forall \tau_i \in \tau} \frac{C_i}{T_i} \tag{3.14}$$

Où τ définit l'ensemble des tâches de la VM ρ^s étudiée.

L'Équation 3.13 se comprend ainsi : pour toutes les VMs, l'utilisation des tâches à l'intérieur des VMs doit être inférieure à 1. Cela signifie donc que la charge processeur de toutes les tâches de toutes les VMs doit être inférieure à 100%.

Si les overheads d'activation des VMs sont également pris en compte, l'Équation 3.13 devient alors :

$$\sum_{\forall \rho^s \in \rho} \left(U^s + \frac{O^s}{T^s} \right) \leq 1 \tag{3.15}$$

Cette équation n'est précise que dans le contexte où les VMs n'ont qu'un seul candidat d'instant critique (une seule exécution non préemptive par période d'activation de la VM). Nous verrons dans le Chapitre 4 à quoi cela nous servira. Elle peut se résumer ainsi : la somme de l'utilisation de toutes les VMs plus l'overhead des VMs doit être inférieur à 100%.

Nous pouvons également l'étendre dans le cas de plusieurs candidats d'instant critique :

$$\sum_{\forall \rho^s \in \rho} \left(U^s + \sum_{\forall \rho^s_{C_k} \in \rho^s_C} \left(\frac{O^s}{T^s} \right) \right) \leq 1 \tag{3.16}$$

Pour les équations suivantes, nous ne les définirons que dans le contexte d'un seul candidat d'instant critique. En effet, elles ne seront utilisées que dans un contexte qui produira un ordonnancement avec un motif constitué d'un seul intervalle d'exécution (ordonnancement strictement périodique des VMs).

Notre but est maintenant de déterminer des bornes pour les paramètres des VMs, avec donc un seul candidat d'instant critique. Ces bornes sont définies avec overhead. Si aucun overhead n'est présent, il suffit alors de fixer ce paramètre à 0. Ces bornes seront utilisées dans le Chapitre 4 pour trouver les paramètres des VMs.

Nous commençons par une borne minimale sur la période des VMs. Cette borne est appelée $T^s_{\beta^1}$ pour la VM ρ^s. Elle se définit ainsi :

$\forall \rho^s \in \rho$,

$$U^s + \frac{O^s}{T^s_{\beta^1}} \leq 1 \tag{3.17}$$

$$\Leftrightarrow \quad \frac{O^s}{T^s_{\beta^1}} \leq 1 - U^s \tag{3.18}$$

$$\Leftrightarrow \quad \frac{O^s \times T^s_{\beta^1}}{T^s_{\beta^1}} \leq (1 - U^s) \times T^s_{\beta^1} \tag{3.19}$$

$$\Leftrightarrow \quad O^s \leq (1 - U^s) \times T^s_{\beta^1} \tag{3.20}$$

$$\Leftrightarrow \quad \frac{O^s}{1 - U^s} \leq T^s_{\beta^1} \tag{3.21}$$

$$\Leftrightarrow \quad T^s_{\beta^1} \geq \frac{O^s}{1 - U^s} \tag{3.22}$$

Pour expliquer le résultat de l'Inéquation (3.22), nous pouvons repartir de l'Inéquation (3.17). Cette équation reprend l'Inéquation (3.15), et l'adapte pour isoler le terme $T^s_{\beta^1}$, qui était dans l'ancienne Inéquation (3.15) T^s.

Pour commencer, nous soustrayons U^s des deux côtés de l'Inéquation (3.18). Le but est maintenant de supprimer du côté gauche de l'inéquation l'élément $T^s_{\beta^1}$. Cela nous donne l'Inéquation (3.19). Cette inéquation est ensuite simplifiée dans l'Inéquation (3.20). À ce moment, l'objectif est maintenant d'isoler le $T^s_{\beta^1}$, en divisant les deux côtés de l'inéquation par $1 - U^s$. Cet élément étant positif, puisque U^s est forcément inférieur ou égal à 1 pour que le système soit faisable, le signe de l'inéquation ne change pas. Nous obtenons alors l'Inéquation (3.21). Nous inversons ensuite les deux éléments de l'inéquation pour finalement obtenir l'Inéquation 3.22 qui décrit une première période minimale pour la VM ρ^s.

Un premier problème se pose avec cette borne : les résultats peuvent être trop pessimistes, surtout en prenant en compte l'overhead. Cela est également dû aux arrondis que nous devons effectuer, les résultats n'étant pas sous forme entière et la période devant l'être. En effet, avec une telle formule, l'overhead peut prendre jusqu'à un maximum de 100% de la période. Ainsi toute la VM n'est alors occupée que par l'overhead et aucune tâche ne peut alors s'exécuter. C'est pour cela que nous ajoutons à celle-ci une autre borne sur la période minimale $T^s_{\beta^2}$ à la VM ρ^s, qui, cette fois, va permettre de borner le pourcentage de temps que l'overhead peut utiliser par rapport à la période de la VM :

$$T^s_{\beta^2} \geq \frac{O^s}{\alpha} \tag{3.23}$$

Où $0 \leq \alpha \leq 1$. En effet, si nous voulons par exemple que l'overhead de 1 représente au maximum 5% de la période, alors $T^s_{\beta^2} \geq \frac{1}{0.05}$, donc $T^s_{\beta^2} \geq 20$. Malgré cela, une troisième borne sera nécessaire. Celle-ci dépendra du nombre de VMs utilisées

dans le système. Par exemple, si nous avons besoin de 4 VMs, il est nécessaire que la période minimale de chaque VM soit au moins de 4. Cela permettra ainsi d'ordonnancer toutes les VMs de manière harmonique, comme nous le verrons dans le Chapitre 4. Nous définissons ainsi une dernière borne : $T^s_{\beta^3}$ pour la VM ρ^s quand le système contient v VMs :

$$T^s_{\beta^3} \geq v \tag{3.24}$$

Où v est le nombre de VMs dans le système. Nous pouvons maintenant obtenir la borne finale sur la période minimale d'une VM dans le système T^s_{min}, qui correspond au maximum des trois bornes précédemment introduites :

$$T^s_{min} \geq \max(\lceil T^s_{\beta^1} \rceil, \lceil T^s_{\beta^2} \rceil, T^s_{\beta^3}) \tag{3.25}$$

Les résultats donnés par les Équations 3.22 et 3.23 pouvant ne pas être entiers, nous devons prendre les valeurs entières immédiatemment supérieures pour ne pas modifier les bornes. Ces bornes étant des bornes sur les périodes minimales, nous ne pouvons alors que considérer l'entier supérieur à cette borne, l'entier inférieur donnant une borne plus large et donc qui peut être fausse. C'est également pour cela qu'un maximum est requis sur les différentes équations de borne minimale pour prendre en compte la contrainte la plus forte. Par exemple, si $\lceil T^s_{\beta^1} \rceil$ donne 1, $\lceil T^s_{\beta^2} \rceil$ donne 3 et que $T^s_{\beta^3}$ donne 4, sélectionner une borne autre que 4 serait en contradiction avec la borne $T^s_{\beta^3}$, celle-ci indiquant que la période minimale doit être de 4 minimum.

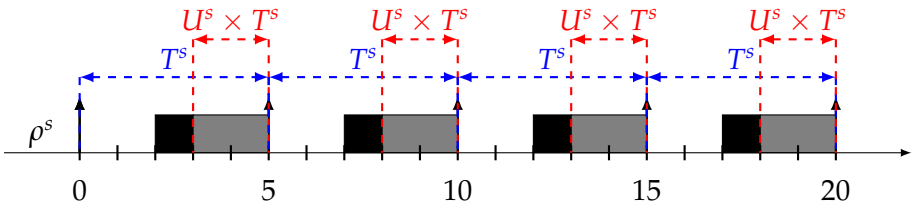


FIGURE 3.17 : Représentation graphique de la borne maximale sur la période de la VM $\rho^1(3, 5, 1)$

Nous allons maintenant proposer une borne maximale sur la période, toujours dans le cas d'un seul candidat d'instant critique. Pour ce faire, nous nous appuyons sur la Figure 3.17. Dans cette figure, nous mettons en évidence, ce à quoi correspond la période T^s de la VM ρ^1 et surtout ce que signifie graphiquement la valeur $U^s \times T^s$. Cette valeur correspond pour chaque période à l'utilisation des tâches de la VM multipliée par la période. La durée d'exécution de la VM n'étant pas encore connue à ce moment, nous partons de l'hypothèse que cette durée est au moins supérieure ou égale à $U^s \times T^s$, sans quoi les tâches de la VM ne peuvent pas être ordonnançables (condition nécéssaire).

Cette borne va également prendre en compte les valeurs minimales des échéances d'un jeu de tâches de la VM. En effet, nous connaissons les candidats d'instant critique d'une VM. C'est sur cela que se base la formule : l'échéance d'une tâche doit permettre que celle-ci soit activée à la fin d'exécution de la VM. Pour un jeu de tâches τ associé à une VM ρ^s, la borne maximale sur la période T^s_{max} est la suivante (condition nécessaire sur l'échéance d'une tâche) :

$\forall \tau_i \in \tau$,

$$D_i \geq T^s_\gamma - U^s \times T^s_\gamma + C_i \tag{3.26}$$

$$\Leftrightarrow \quad D_i - C_i \geq T^s_\gamma \times (1 - U^s) \tag{3.27}$$

$$\Leftrightarrow \quad \frac{D_i - C_i}{(1 - U^s)} \geq T^s_\gamma \tag{3.28}$$

$$\Leftrightarrow \quad T^s_\gamma \leq \frac{D_i - C_i}{(1 - U^s)} \tag{3.29}$$

$$T^s_{max} \leq \left\lfloor \frac{D_i - C_i}{(1 - U^s)} \right\rfloor \tag{3.30}$$

$$\tag{3.31}$$

Pour expliquer ce résultat, partons de l'Équation 3.26. Cette première équation s'explique à l'aide de la Figure 3.17 : les échéances des tâches de la VM doivent être supérieures à la période de la VM moins le début minimal du début de l'exécution de la VM (après overhead s'il est présent) plus le coût de la tâche.

Pour passer à l'Équation 3.27 suivante, le C^i est passé à gauche de l'inéquation et T^s_{max} a été factorisé. Il suffit ensuite d'isoler le T^s_{max} à droite en divisant par $1 - U^s$ pour obtenir l'Équation 3.28. Pour obtenir l'équation finale pour T^s_{max}, il suffit ensuite d'inverser les deux éléments de l'inéquation et ainsi de changer le signe de sens. Nous obtenons ainsi l'Équation (3.30) de laquelle nous déduisons la borne maximale sur la période des VMs en fonction des tâches de chaque VM dans l'Équation (3.31). Dans cette équation, nous ajoutons également un arrondi inférieur, le résultat n'étant pas forcément entier. L'arrondi doit cette fois être inférieur pour s'assurer du respect de la borne maximale trouvée.

Grâce à ces différentes conditions, nous pouvons maintenant déterminer les paramètres des VMs. Il sera alors possible d'après ces paramètres de trouver un ordonnancement pour les VMs.

Chapitre 4

Ordonnancement de VMs hypervisées harmoniques : comment ordonnancer facilement les VMs sans nuire à l'ordonnançabilité ?

Dans ce chapitre, nous étudions comment ordonnancer les VMs avec une périodicité stricte, et donc avec des périodes harmoniques. Pour cela, nous commençons par définir l'harmonicité des VMs sur les périodes deux à deux. Cette définition et les conditions sur notre modèle précédemment définies dans la Section 3.5, nous permettent de déduire les paramètres pour la première VM du système, donc la plus critique. Cette VM a ainsi la plus petite période. Il ne reste plus qu'à trouver les paramètres des autres VMs à partir des paramètres de la première VM (en augmentant les périodes afin qu'elles soient distinctes et harmoniques). Une fois les paramètres déduits, nous présentons l'ordonnancement des VMs à partir des paramètres et de la périodicité stricte. Grâce à cet algorithme, les ordonnancements des VMs ne produisent qu'un seul candidat d'instant critique. Il est ainsi possible d'utiliser l'analyse de temps de réponse pour les tâches présentée dans la Section 3.3.

Sommaire

4.1 Harmonicité

Pour commencer, nous rappelons ce qu'est la propriété d'harmonicité contrainte. Pour des raisons de simplicité, nous utiliserons le terme harmonicité pour décrire l'harmonicité contrainte. Nous utilisons cette propriété au cours de ce chapitre pour établir les conditions d'ordonnançabilité de VMs strictement périodiques.

Définition 45 (Harmonicité contrainte)
Des tâches sont dites harmoniques lorsque toutes leurs périodes sont multiples deux à deux. Ainsi, des VMs sont harmoniques lorsque toutes les périodes des VMs sont multiples deux à deux.

Pour donner un exemple, nous pouvons parler de quatre tâches (ou VMs) harmoniques lorsque leurs périodes sont respectivement 2, 4, 8 et 16.

Cette propriété nous servira par la suite à ordonnancer les VMs grâce à des algorithmes ne fonctionnant que lorsque les VMs ont des périodes harmoniques.

Dans l'état de l'art concernant l'ordonnancement mono-processeur de tâches harmoniques, il existe une condition d'ordonnançabilité nécessaire et suffisante pour que toutes les tâches respectent leurs échéances. Il faut également ajouter que cette condition n'est valable que lorsque les tâches ont des périodes distinctes. Elle a été définie dans [52] (Section 2.1) et est reprise dans le théorème suivant :

Théorème 2 (Condition d'ordonnançabilité d'un jeu de tâches harmonique avec des périodes distinctes)
Prenons τ un jeu de tâches harmoniques tel que $\forall \tau_i \in \{\tau \setminus \{\tau_1\}\}, T_i > T_{i-1}$. Les tâches du jeu de tâches τ respectent toutes leurs échéances si et seulement si :

$$\forall \tau_i \in \{\tau \setminus \{\tau_1\}\}, C_i \leq T_1 - C_1 \tag{4.1}$$

Nous adaptons ainsi ce théorème à notre contexte d'ordonnancement hiérachique en l'appliquant sur les VMs avec overhead :

Théorème 3 (Condition d'ordonnançabilité de VMs harmoniques avec des périodes distinctes)
Prenons ρ un ensemble de VMs harmoniques tel que $\forall \rho^s \in \{\rho \setminus \{\rho^1\}\}, T^i > T^{i-1}$. L'ensemble des VMs est ordonnançable si et seulement si :

$$\forall \rho^s \in \{\rho \setminus \{\rho^1\}\}, U^s \times T^s + O^s \leq T^1 - C^1 \tag{4.2}$$

Les changements sont les suivants : le Worst Case Execution Time (WCET) C_i d'une tâche devient alors pour une VM ρ^s, $U^s \times T^s$, $U^s \times T^s$ étant la période d'activation de la VM ρ^s pendant lequel une VM doit être exécutée pour ordonnancer les tâches de la VM. Nous ajoutons à cela l'overhead d'activation de la VM, à savoir, O^s. La partie droite de l'inéquation est une adaptation simple de la période d'une tâche à la période d'une VM et du WCET d'une tâche au temps d'exécution de la VM.

4.2 Déduction des paramètres de la VM la plus critique

Dans cette section, nous proposons un algorithme pour déduire les paramètres de la première VM du système, la plus critique. Pour cela, nous utilisons les bornes minimales et maximales sur la période de la VM, déterminées dans la Section 3.5 et l'analyse de temps de réponse des tâches dans une VM de la Section 3.3.

Indice :	0	1	2	3	4	5	6	7	8
Valeur :	0	3	5	8	10	12	14	16	18

TABLE 4.1 : Tableau classé pour une recherche dichotomique

Indice :	0	1	2	3	4	5	6	7	8
Valeur :	0	3	5	8	10	12	14	16	18
Valeur :	0	3	5	8	10	12	14	16	18
Valeur :	0	3	5	8	10	12	14	16	18
Valeur :	0	3	5	8	10	12	14	16	18
Valeur :	0	3	5	8	10	12	14	16	18

TABLE 4.2 : Résultats de la recherche dichotomique

Pour ce faire, nous utilisons un algorithme qui effectue une recherche dichotomique pour trouver la plus petite période possible pour la première VM.

Commençons par donner un exemple de recherche dichotomique. Prenons le tableau de la Table 4.1. Nous avons ici 9 éléments classés par ordre croissant et répartis dans un tableau jusqu'à un indice de 8. Si nous souhaitons trouver l'indice de l'élément 18, nous allons d'abord tester l'élément médian : $\frac{8}{2} = 4$. L'élément d'indice 4 du tableau est ainsi 10. Comme $18 \geq 10$, nous allons recommencer sur la partie de droite du tableau. Cette fois, $\frac{8+4}{2} = 6$, nous allons donc tester l'élément d'indice 6, soit 14. Comme $18 \geq 14$, nous allons recommencer avec la partie droite du tableau : $\frac{8+6}{2} = 7$. L'élément d'indice 7 étant 16, nous testons $18 \geq 16$. N'ayant plus qu'un seul élément à tester, $18 = 18$, l'indice de l'élément 18 est donc 8.

Nous montrons dans la Table 4.2 un visuel du calcul de la recherche dichotomique. En effet, le premier élément testé est 10, celui-ci doit alors être comparé à l'élément recherché, 18. Le second élément est ensuite 14, suivi de 16, avant de finalement tester le dernier élément, 18. La complexité de cet algorithme de recherche dichotomique est de $\log_2(n)$ pour une liste à n éléments, ce qui en fait un des algorithmes de recherche les plus efficaces.

Nous décrivons, dans l'Algorithme 2, l'algorithme pour définir les paramètres de la première VM. Comme expliqué précédemment, cet algorithme utilise une recherche dichotomique. Cette recherche est faite dans la boucle "tant que" (Ligne 3). La recherche dichotomique se fait ainsi : nous recherchons la plus petite période telle que le jeu de tâches de la VM soit ordonnançable, en sachant que la période est bornée par T^1_{min} et T^1_{max}.

L'algorithme prend en entrée T^1_{min}, T^1_{max} et τ, respectivement la borne minimale sur la période de la VM ρ^1, la borne maximale et le jeu de tâches τ associé à ρ^1. Les deux premières lignes de l'algorithme sont des initialisations : nous initialisons T^1 à la valeur T^1_{min} (Ligne 1) et la variable *sched* (Ligne 2), qui est une variable booléenne à $faux$.

Dans la boucle, nous commençons par tester une valeur pour le temps d'exécution de la VM C^1 (Ligne 4). Grâce à cette valeur, nous avons des valeurs à tester pour C^1 et T^1, ce qui représente tous les paramètres de la VM ρ^1. Nous pouvons alors tester si les tâches contenues dans τ respectent leurs échéances (Ligne 5), d'après les Équations (3.1) et (3.7), si la VM a, ou n'a pas, d'overhead.

Si toutes les tâches sont ordonnançables, la variable booléenne *sched* passe alors à *vrai* (Ligne 6), sinon elle passe à $faux$ (Ligne 8).

Le test suivant (Ligne 9) permet de savoir si un ordonnancement a été trouvé : si toutes les tâches respectent leurs échéances et que la valeur de T^1 testée est égale à

Algorithme 2 : Paramètres de la VM la plus critique

Entrées : $(T^1_{min}, T^1_{max}, \tau)$
Sorties : (C^1, T^1, sched)

```
1  T^1 = T^1_min;
2  sched = faux;
3  tant que T^1_min + 1 ≠ T^1_max faire
4      C^1 = ⌈U^1 × T^1 + O^1⌉;
5      si τ est ordonnançable d'après (3.1) (ou (3.7)) alors
6          sched = vrai;
7      sinon
8          sched = faux;
9      si sched = vrai and T^1 = T^1_min alors
10         retourner (C^1, T^1, sched);
11     sinon si sched = vrai alors
12         T^1_max = T^1;
13     sinon
14         T^1_min = T^1;
15     T^1 = ⌊(T^1_min + T^1_max)/2⌋;
16 retourner (C^1, T^1, sched);
```

T^1_{min}, alors les paramètres de la VM ρ^1 sont trouvés et retournés (Ligne 10).

La ligne suivante permet de tester si les tâches respectent leurs échéances mais que $T^1 != T^1_{min}$ (Ligne 11). Ainsi, nous changeons la borne de T^1_{max} (Ligne 12) pour que la recherche dichotomique puisse être effectuée sur une zone plus restreinte.

La ligne suivante correspond au cas où les tâches n'ont pas pu respecter leurs échéances (Ligne 13). Dans ce cas, la valeur de T^1_{min} est changée et devient celle de T^1 (Ligne 14). Cela permet ainsi de tester la recherche dichotomique sur des périodes plus grandes.

La ligne suivante change la valeur de la période T^1 testée. En effet, si l'algorithme n'a pas encore trouvé de solution, il va devoir tester une nouvelle valeur pour T^1 et recommencer, comme fait dans une recherche dichotomique (Ligne 15).

La dernière ligne (Ligne 16) permet d'avoir un retour sur l'algorithme. En effet, plusieurs cas sont alors possibles : soit l'algorithme a trouvé un résultat positif c'est-à-dire des valeurs possibles pour T^1 et C^1 où les tâches respectent leurs échéances, soit elles ne les respectent pas. Ces deux cas pourront être dissociés grâce à la variable booléenne *sched*, elle-même retournée par l'algorithme en plus des dernières valeurs testées. Cela permet de tester le cas où l'algorithme a trouvé des valeurs ordonnançables, mais que T^1 n'était pas égal à T^1_{min}, et que des valeurs intermédiaires ne sont plus possibles à trouver.

Ainsi, nous déduisons grâce à l'Algorithme 2 la plus petite période T^1 et le temps d'exécution C^1 associé de la VM ρ^1 pour lesquels toutes les tâches respectent leurs échéances. Une période petite permet d'ordonnancer plus de VMs qu'une période plus grande, comme nous le verrons dans la Section 4.3, nous utilisons donc la plus petite période possible.

4.3 Déduction des paramètres des autres VMs par ordre de criticité décroissante

Dans cette section, nous allons maintenant voir comment déterminer les paramètres C^s et T^s des autres VMs ρ^s que ρ^1, $\rho^s \in \{\rho \setminus \rho^1\}$.

Toutes les VMs ne pourront pas forcément obtenir des paramètres grâce à ces algorithmes. D'autres méthodes sont proposées dans le Chapitre 5 pour le cas des VMs sans paramètres définis, en relâchant les contraintes de périodicité stricte.

Nous considérons donc ici que les paramètres de la première VM ρ^1 sont trouvés et connus. Ces paramètres seront la base des algorithmes pour trouver les paramètres des VMs suivantes. En effet, nous cherchons ici à trouver des paramètres où les périodes des tâches sont harmoniques. Ainsi, les paramètres que nous souhaitons obtenir dépendent de ceux déjà trouvés. L'algorithme sera ici itératif : il cherche d'abord des paramètres pour la seconde VM ρ^2, puis la troisième VM ρ^3 et ainsi de suite. Toutes les VMs sont ainsi testées, et les VMs dont aucuns paramètres n'a pu être trouvé seront traitées dans le Chapitre 5. Nous visons ainsi à maximiser le nombre de VMs ordonnancées de manière harmonique.

Algorithme 3 : Paramètres des VMs harmoniques

Entrées : ρ
Sorties : listeVM

```
1  listeVM = liste();
2  indiceVM = 1;
3  pour chaque ρ^s ∈ {ρ \ ρ^1} faire
4      T^s_max = (3.31);
5      T^s_test = T^indiceVM × 2;
6      si T^s_test ≤ T^s_max alors
7          C^s_test = ⌈U^s × T^s_test + O^s⌉;
8          si C^s_test ≤ T^1 − C^1 et τ est ordonnançable d'après (3.7) alors
9              C^s = C^s_test;
10             T^s = T^s_test;
11             listeVM += ρ^s(C^s, T^s);
12             indiceVM = s;
13         sinon
14             continuer;
15 retourner listeVM;
```

Nous montrons dans l'Algorithme 3 l'algorithme utilisé pour définir les paramètres, si possible, des autres VMs. Tout d'abord, cet algorithme prend en entrée la liste des différentes VMs du système ρ. Dans cette liste, seule ρ^1 a ses paramètres définis, selon la Section 4.2. À ce stade, les autres VMs n'ont pas leur paramètres définis (durée, période), seuls les jeux de tâches devant s'exécuter dans la VM sont définis. Le retour de cet algorithme est une liste de VMs avec paramètres. Toutes les VMs ne feront pas forcément partie de cette liste si les paramètres ne respectent pas les bornes sur la période ou si les tâches ne peuvent pas respecter leurs échéances avec de tels paramètres.

Le retour de l'algorithme est la liste des VMs sur laquelle les paramètres ont été trouvés. Cette liste est ensuite initialisée à la Ligne 1. Une autre initialisation est faite Ligne 2. Il s'agit de l'indice de la dernière VM pour laquelle les paramètres ont été

trouvés. Cette variable sera utilisée par la suite pour trouver une période, la plus petite possible pour la nouvelle VM.

L'algorithme se poursuit ensuite avec une boucle Ligne 3. Cette boucle se fait pour toutes les VMs, sauf la première. En effet, nous cherchons ici à déterminer les paramètres des VMs du système, sauf la première, puisqu'ils ont déjà été obtenus. Ainsi, pour chaque VM à tester du système, nous déterminons une borne sur la période maximale T^s_{max} à l'aide de l'Équation (3.7) (ou (3.1) dans le cas où il n'y a pas d'overhead). Cette équation prend également en compte le jeu de tâches associé à la VM testée, celui-ci doit donc être connu.

Nous testons ensuite une valeur pour la période de la VM. Cette variable de test est appelée T^s_{test} et est initialisée Ligne 5. La valeur que nous donnons ici est la dernière période trouvée, soit $T^{indiceVM}$ multipliée par 2. Cela nous permet de nous assurer que toutes les périodes résultantes sont harmoniques entre elles, distinctes et de s'assurer qu'elles restent les plus petites possibles. Puisque les périodes maximales des différentes VMs sont rapidement atteintes, cette multiplication par deux permet de s'assurer que le plus de VMs possible n'atteint pas la période maximale.

Pour donner un exemple, si nous prenons un système constitué de quatre VMs et que $\rho^1(1,3)$, alors les périodes de test pour ρ^2, ρ^3 et ρ^4 sont respectivement 6, 12, 24, ce qui donne des périodes harmoniques entre elles.

L'étape suivante de l'algorithme est de tester si la période de test est plus petite ou égale à la borne maximale sur la période (Ligne 6). Si cette étape est respectée, alors nous créons une valeur de test pour le temps d'exécution de la VM. Cette étape est faite Ligne 7. La valeur choisie ici est $\lceil U^s \times T^s_{test} + C^s_o \rceil$. Cela correspond donc à l'utilisation du jeu de tâches de la VM multiplié par la période de la VM. Nous l'avions déjà indiqué dans la Figure 3.17, cela représente le temps d'exécution minimum qu'il faut à la VM pour exécuter le jeu de tâches présent. Nous ajoutons à cela l'overhead de la VM pour s'assurer qu'il soit pris en compte et que les tâches aient le temps de s'exécuter. Nous aurions pu choisir un C^s_{test} plus grand, mais les résultats sont bons. Un plus grand temps d'exécution de VMs auraient peut-être permis d'ordonnancer plus de jeux de tâches, mais cela aurait eu un impact négatif sur le nombre de VMs ordonnançables de manière harmonique, le temps d'exécution des VMs étant borné.

L'étape suivante (Ligne 8) de l'algorithme est de tester deux éléments :

- Si le temps d'exécution testé est bien inférieur ou égal à $T^1 - C^1$, comme indiqué par la condition nécessaire et suffisante d'ordonnançabilité des VMs harmoniques, définie dans le Théorème 3 et son Équation (4.2) associée.
- Si le jeu de tâches associé à la VM testé est ordonnançable avec ces paramètres testés, à savoir C^s_{test} et T^s_{test} et d'après l'Équation (3.7).

Si ces deux conditions sont réunies, les paramètres testés pour la VM sont alors valables. Nous validons alors les paramètres testés, à savoir le temps d'exécution de la VM C^s (Ligne 9) et sa période T^s (Ligne 10). Nous ajoutons ensuite la VM et ses paramètres validés à la liste de VMs pour laquelle nous avons trouvé les paramètres (Ligne 11). Nous actualisons ensuite l'indice de la dernière VM (Ligne 12) pour laquelle nous avons trouvé des paramètres pour que les nouvelles périodes testées soient les bonnes.

Si une des deux conditions précédentes n'est pas respectée, nous passons alors à la VM suivante (Ligne 14). Pour finir, une fois toutes les VMs testées, l'algorithme retourne la liste des VMs pour lesquelles des paramètres ont été trouvés (Ligne 15).

Les VMs dont les paramètres n'ont pu être attribués sont traitées dans le Chapitre 5.

4.4 Ordonnancement harmonique des VMs

Dans cette section, nous montrons comment ordonnancer des VM de manière harmonique. L'algorithme ici présenté est une adaptation de l'algorithme présenté dans [52]. Nous reprenons de ce papier l'algorithme pour ordonnancer des tâches avec des périodes harmoniques. Le contexte ici utilisé est celui de tâches harmoniques activées de manière synchrone et non-préemptive. Nous utiliserons les propriétés de l'harmonicité pour ordonnancer les tâches le plus simplement possible. Pour cela, nous utilisons des approches générales. Il s'agit du Bin-Packing, et de l'algorithme First-Fit Decreasing.

Définition 46 (Bin-Packing)
Les problèmes de Bin-Packing sont un type de problèmes de complexité NP-Difficile. Les problèmes NP-Difficiles sont une classe particulière de problèmes de décision pour lesquels une solution peut être vérifiée en temps polynomial, mais trouver cette solution revient à un problème de classe NP. Ces problèmes consistent en ranger un nombre fini d'objets de différents volumes dans différentes boîtes, appelées bins, d'une dimension ou contenance donnée.

Définition 47 (Algorithme First-Fit Descreasing)
L'algorithme First-Fit Decreasing permet de placer les objets classés dans un ordre décroissant, selon un paramètre, dans la première boîte ou bin dans laquelle l'objet tient. Il s'agit d'une heuristique pour répondre au problème de Bin-Packing.

Dans le domaine de l'ordonnancement temps réel, ces différents algorithmes sont utilisés dans un contexte multi-processeurs où chaque processeur correspond à une boîte, et les tâches correspondent aux objets. Ici, nous allons l'utiliser dans un contexte mono-processeur grâce aux propriétés des tâches harmoniques. En effet, cette fois, la boîte n'est plus un processeur mais une fenêtre temporelle dans laquelle placer les différentes durées d'exécution des tâches dont le nombre dépend de la période des tâches. Puisque les tâches sont harmoniques entre-elles, leur hyper-période est égale à la plus grande période des tâches. L'hyper-période correspond à la somme des fenêtres temporelles de l'ensemble des boîtes, qui est séparée en $\frac{T_m}{T_1}$ boîtes de taille T_1, où m est le nombre de tâches et les tâches sont classées par ordre décroissant des inverses des périodes. Les objets restent les tâches. Le but est maintenant de placer les différentes tâches classées. La première tâche sera ainsi placée en premier dans chaque boîte, et les autres tâches seront placées dans les premières boîtes qui peuvent les contenir si la tâche est bien active. Grâce aux propriétés de l'harmonicité et du respect de conditions telle que Équation (4.1), nous pouvons nous assurer que toutes les tâches sont ordonnancées.

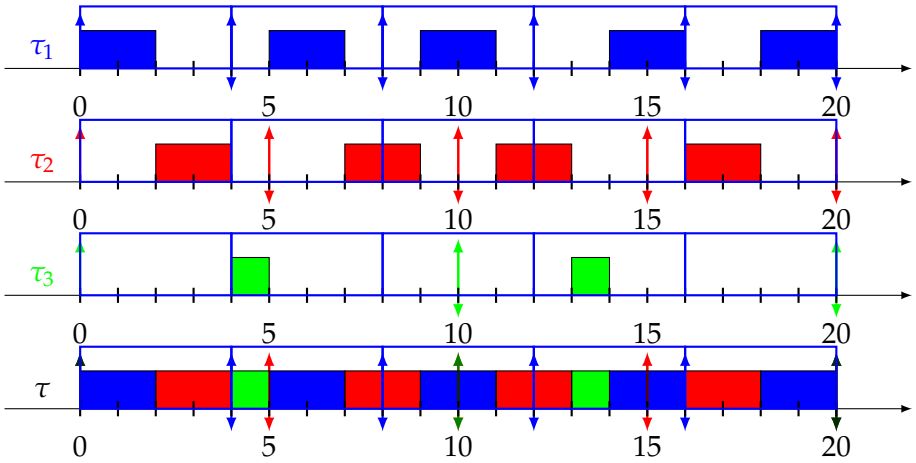


FIGURE 4.1 : Exemple d'ordonnancement First-Fit sans harmonicité avec comme jeux de tâches : $\tau_1(2,4,4)$, $\tau_2(2,5,5)$ et $\tau_3(1,10,10)$

Nous donnons dans la Figure 4.1 un exemple d'ordonnancement avec First-Fit sans harmonicité. Ici, le jeu de tâches à ordonnancer est $\tau_1(2,4,4)$, $\tau_2(2,5,5)$ et $\tau_3(1,10,10)$. Nous considérons ici un First-Fit simple : il contient une liste d'instances de tâches à exécuter et exécute la première dans la première boîte disponible. La liste d'instances de tâches est ainsi actualisée à chaque nouvelle activation de tâches et chaque nouvelle tâche est insérée en fin de liste.

Pour donner un exemple sur les premières exécutions, à l'instant 0, la liste est : τ_1, τ_2 et τ_3. Les tâches τ_1 et τ_2 sont ainsi exécutées. À l'instant 4, une nouvelle activation de τ_1 se fait, la liste contient ainsi : τ_3 et τ_1. La tâche τ_3 est donc d'abord exécuté. À l'instant 5, une nouvelle activation de τ_2 se fait, la liste contient alors τ_1 et τ_2. La tâche τ_1 est donc d'abord exécutée, suivie de τ_2. Les exécutions se poursuivent ainsi jusqu'à la fin.

Pour revenir sur l'algorithme présenté dans [52], il utilise le First-Fit Decreasing, mais avec une autre approche. Les tâches sont ici ordonnancées dans l'ordre décroissant de l'inverse de la période, et chaque tâche est ordonnancée dans la première boîte libre, et donc où aucune autre tâche n'est ordonnancée. L'algorithme n'est pas détaillé dans le papier, c'est pourquoi nous en donnons un algorithme formel et adapté à notre modèle dans les Algorithmes 4 et 5.

Commençons par l'Algorithme 4. Ses entrées sont la VM ρ^1 avec ses paramètres et un sous ensemble de VMs, avec des paramètres trouvés grâce à l'Algorithme 3, classé par périodes croissantes (comme le calcule l'Algorithme 4). Avec ces entrées, le but est maintenant d'obtenir en sortie les ordonnancements des VMs. Cela se fait grâce à la variable $ordoVMs$ qui correspond à une liste de listes de débuts et de fins des exécutions des VMs harmoniques. C'est donc dans cette variable que seront enregistrés tous les ordonnancements des VMs, y compris la première, ρ^1.

Commençons par les initialisations avant l'algorithme à proprement parler. Ligne 1, nous initialisons la liste qui contiendra la liste des débuts et des fins des exécutions des VMs. Ligne 2, nous initialisons la variable n comme étant égale à la taille du sous-ensemble de VMs avec paramètres, pour lesquelles nous devons trouver un ordonnancement auquel nous ajoutons 1 de manière à extraire le dernier élément de cette liste facilement. Ligne 3, nous initialisons la variable $repetitionsVM1$. Cette variable correspond au nombre d'activations que la première VM ρ^1 aura lors de l'ordonnancement avant de se répéter. Les VMs étant harmoniques, la répétition de

Algorithme 4 : Algorithme d'ordonnancement de VMs harmoniques

Entrées : ($\rho^1(C^1, T^1, O^1)$, $\rho_{sub} \in \{\rho \setminus \rho^1\}$ le sous-ensemble de VMs avec paramètres)
Sorties : *ordoVMs* : Liste de liste de débuts et de fins d'exécution des VMs

```
1  ordoVMs = liste();
2  n = taille(ρ_sub) + 1;
3  repetitionsVM1 = T^n / T^1;
4  booléen libre[repetitionsVM1] = vrai;
5  ordoVM1 = liste();
6  pour i = 0; i < repetitionsVM1; i = i + 1; faire
7      débutVM1 = i × T^1;
8      finVM1 = i × T^1 + C^1;
9      ordoVM1 += (débutVM1, finVM1);
10 ordoVMs += ordoVM1;
11 pour chaque ρ^j ∈ ρ_sub faire
12     repetitionsVM = T^n / T^j;
13     ordoVM = liste();
14     pour i = 0; i < repetitionsVM; i = i + 1; faire
15         indice = (i × T^j) / T^1;
16         premierLibre = FirstFit(libre, indice, n);
17         libre[premierLibre] = faux;
18         débutVM = premierLibre × T^1 + C^1;
19         finVM = premierLibre × T^1 + C^1 + C^j;
20         ordoVM += (débutVM, finVM);
21     ordoVMs += ordoVM;
22 retourner ordoVMs
```

Algorithme 5 : FirstFit

Entrées : (*libre*[] un tableau booléen et *indice*, l'indice de début de la recherche dans le tableau booléen)
Sorties : L'indice du premier espace libre disponible dans le tableau booléen à partir de l'indice donné en entrée

```
1 pour chaque libre[i] ∈ libre faire
2     si i > indice et libre[i] = vrai alors
3         retourner i;
4     fin
5 fin
```

l'ordonnancement se fait sur la période la plus grande, soit la dernière VM. Nous parlons alors d'hyper-période.

Définition 48 (Hyper-période)
Dans un système temps réel à activation synchrone, l'hyper-période correspond au temps avant lequel l'ordonnancement se répète dans le temps. Pour le trouver, dans le cas de tâches, il faut alors calculer le plus petit commun multiple des périodes des tâches. Pour le cas des VMs uniquement, il s'agit donc du plus petit commun multiple entre les périodes des VMs. Dans le cas de l'harmonicité, puisque toutes les périodes sont multiples entre elles, le plus petit commun multiple correspondant à l'hyper-période est donc la plus grande période.

Le nombre d'activations de la VM ρ^1 est donc de $\frac{T^s}{T^1}$. La Ligne 4 permet d'initialiser un tableau de booléens ayant pour taille le nombre d'activations de la VM ρ^1. Toutes les valeurs du tableau sont alors à *vrai*. Ce tableau permet à l'algorithme First-Fit Decreasing de trouver la première boîte libre où aucune VM n'est ordonnancée (en dehors de la VM ρ^1). Il sera donc actualisé au fur et à mesure quand des boîtes seront réservées pour l'exécution d'une VM.

La Ligne 5 initialise, quant à elle, la liste des ordonnancements de la première VM ρ^1. Cette liste contiendra ainsi tous les débuts et toutes les fins des exécutions de la VM.

La boucle suivante (Ligne 6) se fait pour toutes les activations de la VM ρ^1. C'est dans cette boucle que nous commencerons par déterminer tous les débuts et toutes les fins des exécutions de la VM ρ^1. Ainsi, la Ligne 7 définit le début des exécutions de ρ^1 et la Ligne 8 sa fin. Nous ajoutons ensuite ces valeurs dans la liste $ordoVM1$ (Ligne 9).

Les valeurs choisies pour les débuts et les fins des exécutions correspondent au numéro de l'activation de la VM (en commençant par 0), multipliée par la période de la VM. Ainsi, pour la première activation, le début est à 0, pour la seconde à T^1 et ainsi de suite. La valeur de fin des exécutions est la même à laquelle nous ajoutons la durée de l'exécution.

La boucle suivante, Ligne 11, est la boucle dans laquelle nous définirons les ordonnancements des autres VMs pour lesquelles nous avons pu obtenir des paramètres harmoniques. Le procédé est le même que pour la première VM.

Nous commençons donc par déterminer le nombre d'activations de la VM ρ^j étudiée (Ligne 12) et à initialiser une liste qui contiendra les ordonnancements de la VM (Ligne 13).

La boucle suivante (Ligne 14) reprend le même principe que la première boucle (Ligne 6). Nous allons ainsi déterminer les exécutions de la VM ρ^j étudiée pour chaque activation. Nous commençons par déterminer un indice. Cet indice correspond au numéro d'activation minimum de la VM ρ^1 qui contient une boîte libre pour l'exécution la VM ρ^j.

Lors de la ligne suivante (Ligne 16), nous appelons l'Algorithme 5. Nous définissons ainsi dans la variable *premierLibre* la première activation de la VM ρ^1 pour laquelle aucune autre VM n'est exécutée. Nous indiquons ensuite que cette activation sera utilisée par une VM Ligne 17. Les trois lignes suivantes reprennent le même principe que pour la première VM : nous déterminons le début d'activation de la VM ρ^j (Ligne 18), puis sa fin (Ligne 19) et nous ajoutons les résultats à la liste des débuts et des fins des exécutions de la VM (Ligne 20). Nous ajoutons ensuite cette liste à la liste $ordoVMs$ (Ligne 21) avant de la retourner (Ligne 22).

Pour décrire le dernier Algorithme 5, il reprend le principe de l'algorithme First-Fit vu précédemment. Ses entrées sont donc : le tableau de booléens *libre* créé lors de l'Algorithme 4, pour trouver la première boîte disponible pour ordonnancer la VM,

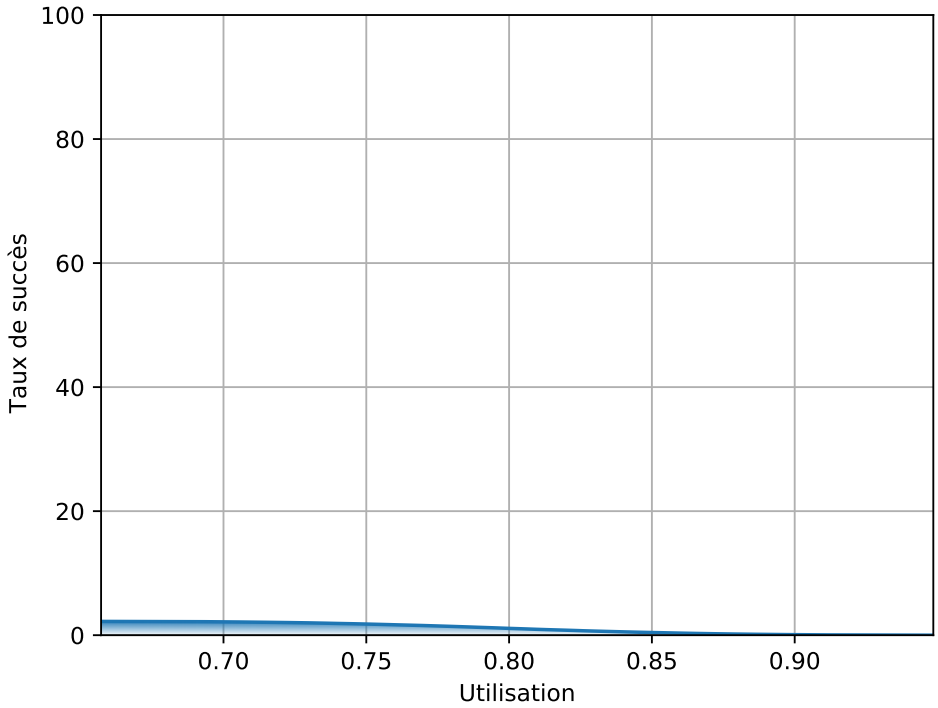


FIGURE 4.2 : Taux de succès d'une seule VM en fonction de l'utilisation

l'indice de début de recherche, puisque celui-ci doit être appelé à plusieurs moments de l'algorithme et selon la période de la VM demandée.

Cet algorithme ressort ensuite l'indice du premier espace disponible dans le tableau booléen à partir de l'indice donné en entrée. Cela correspond donc à la première boîte où aucune autre VM n'est ordonnancée, ainsi, au premier emplacement libre dans l'ordonnancement actuel pour pouvoir ordonnancer une nouvelle VM. L'algorithme est ensuite simple : il teste pour chaque élément du tableau de booléens (Ligne 1), si l'indice est un indice supérieur à l'indice demandé et si l'élément du tableau testé est disponible (Ligne 2). Si tel est le cas, nous retournons alors l'indice de l'élément libre (Ligne 3).

Nous sommes assurés de toujours trouver un indice de retour avec cet algorithme grâce au Théorème 3 et à l'Équation (4.2). Ainsi, les deux algorithmes exposés dans cette section ont un retour sûr, ces équations étant testées précédemment pour trouver les paramètres.

Pour conclure cette section, nous présentons les différents résultats de l'algorithme utilisant l'harmonicité dans des simulations où quatre jeux de tâches sont utilisés dans quatre VMs différentes. Les jeux de tâches sont ici générés de manière aléatoire à l'aide d'algorithmes présentés dans la Section 7.1. L'algorithme utilisé pour les simulations est présenté dans la Section 7.2. Les résultats sont présentés à travers les Figures 4.2, 4.3 et 4.4 pour le cas sans overhead et les Figures 4.5 et 4.6 pour le cas avec overhead.

Dans ces différentes figures, nous présentons le taux de succès de l'algorithme harmonique en fonction de l'utilisation. L'utilisation débute à environ 0.63 et croît jusqu'à se rapprocher de 1. Dans un premier temps, lorsque strictement une seule

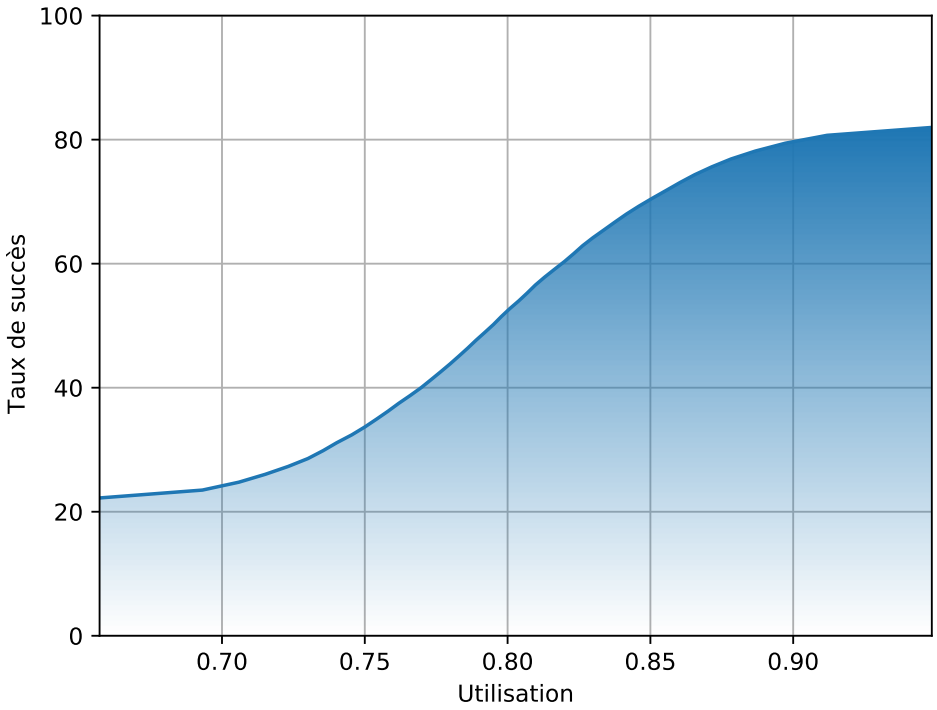


FIGURE 4.3 : Taux de succès d'exactement deux VMs en fonction de l'utilisation

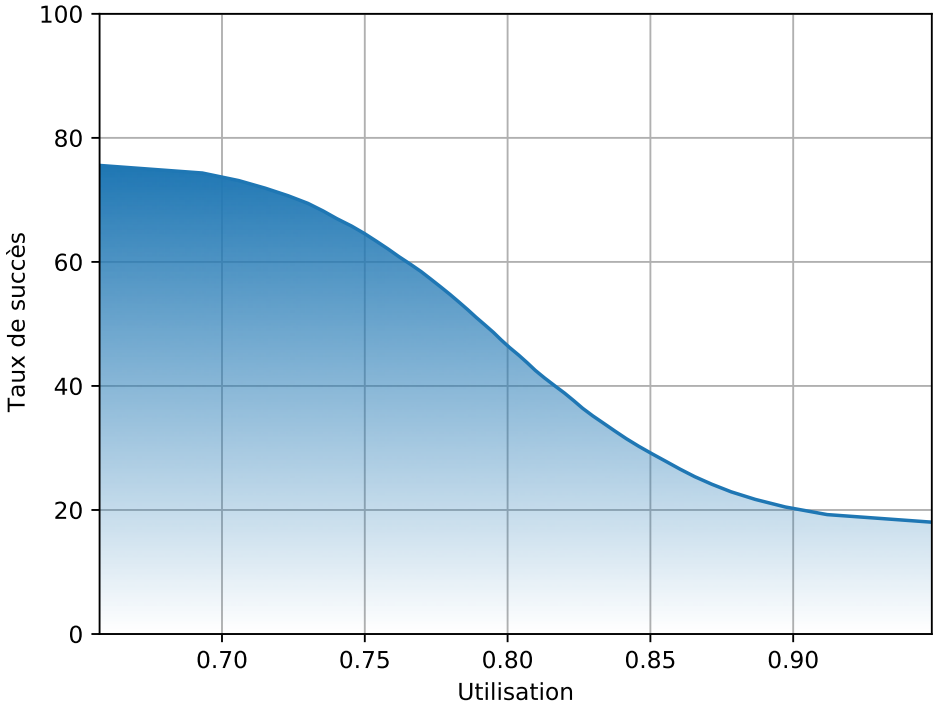


FIGURE 4.4 : Taux de succès d'exactement trois VMs en fonction de l'utilisation

VM est ordonnançable par l'algorithme harmonique (Figure 4.2), puis deux (Figure 4.3) et enfin exactement trois (Figure 4.4). Dans le cas où strictement une seule VM est ordonnançable, donc seulement la première, les résultats sont faibles. Ainsi, elle débute à un taux de succès d'un peu plus de 2% et décroit jusqu'à 0%. Ce résultat était attendu : il est plus courant d'obtenir plusieurs VMs ordonnançables avec l'algorithme harmonique.

Dans le cas de deux VMs ordonnançables (Figure 4.3), les résultats sont bien plus élevés. Cette fois-ci, la courbe croît de 22% des jeux de tâches ordonnancés avec uniquement 2 VMs jusqu'à atteindre un peu plus de 80% en s'approchant d'une utilisation de 1. Dans le cas de trois VMs ordonnancées, le taux de succès décroît de 75% pour une utilisation faible jusqu'à atteindre 0% pour une utilisation proche de 1.

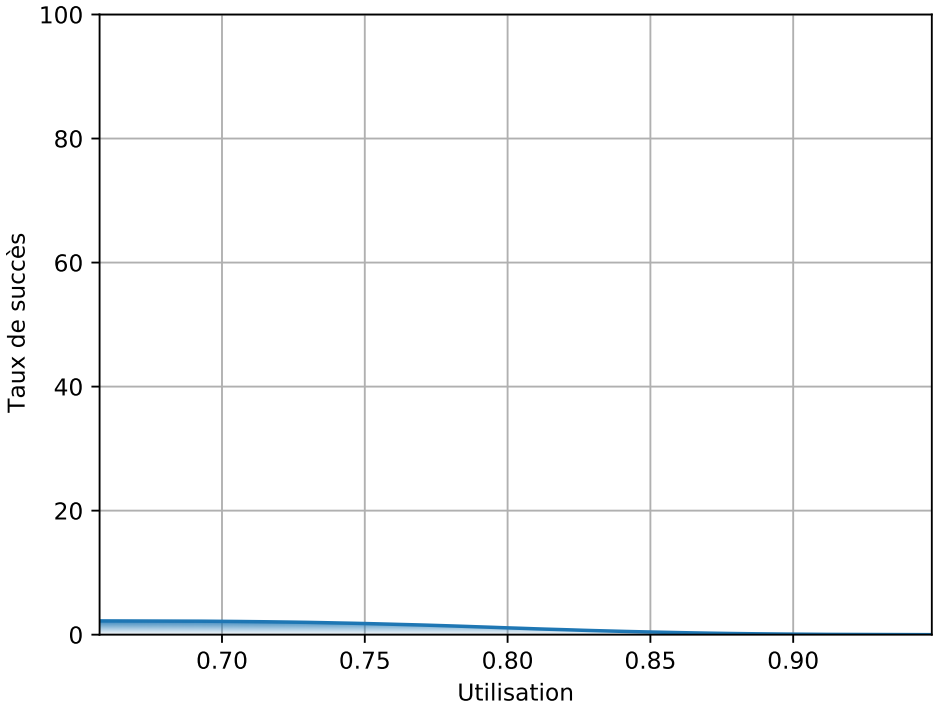
Le taux de succès pour quatre VMs correspondrait quant à lui aux résultats finaux pour l'algorithme harmonique, mais ces résultats ne seront pas présentés ici. En effet, le taux de succès de cet algorithme pour les quatre VMs reste ici constant à 0% des VMs qui sont toutes ordonnancées grâce à l'harmonicité.

Concernant les taux de succès avec overhead, commençons par la Figure 4.5. Dans cette figure, nous observons une courbe décroissante partant d'un taux de succès de 55% allant jusqu'à 15% pour une utilisation de 0.63 à proche de 1. Précédemment, nous ne trouvions que peu de systèmes dans lesquels une seule VM était ordonnancée (décroissance partant d'un peu plus de 2% jusqu'à 0%), nous observons donc une augmentation de plus de 50%. Cela s'explique simplement, du fait des overheads, le système étant plus chargé, donc de moins en moins de VMs peuvent être ordonnancées. Nous obtenons donc beaucoup plus souvent le cas où une seule VM est ordonnancée, au lieu d'une simple moyenne à 5%.

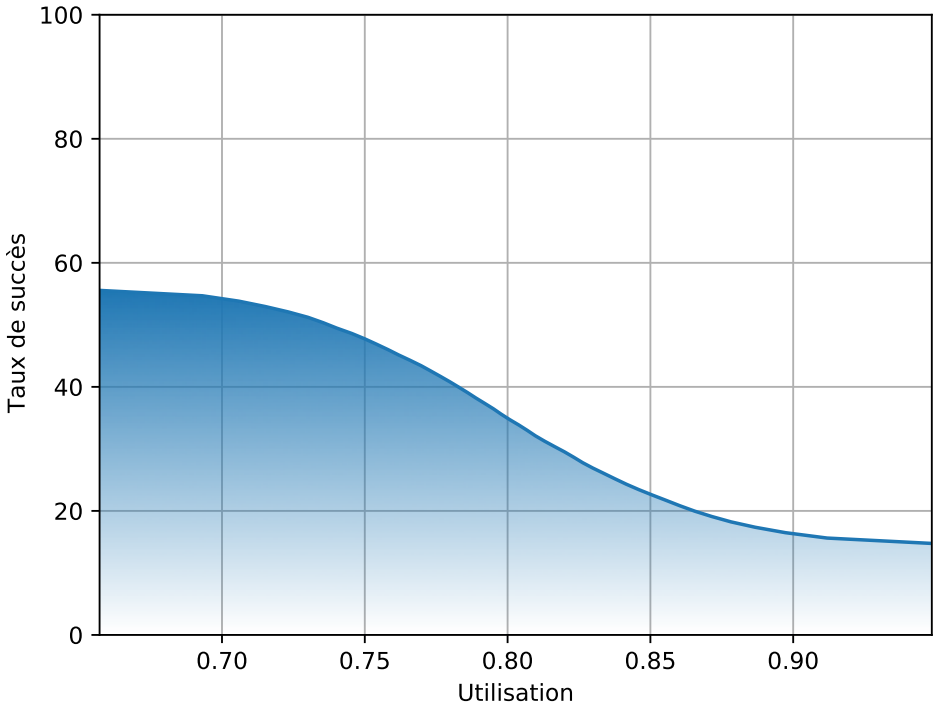
Dans la seconde figure, Figure 4.6, nous observons cette fois une courbe décroissante partant de plus de 3% de taux de succès jusqu'à 0% pour une utilisation allant de 0.63 à 1. Nous obtenons donc un bien moins grand nombre de cas où exactement deux VMs sont ordonnancées de manière harmonique et dont toutes les tâches respectent leurs échéances avec la priorité fixe. Cela s'explique par l'augmentation du nombre de systèmes qui ne fonctionnent qu'à une seule VM. De plus, cette courbe ressemble fortement à celle d'une seule VM ordonnancée de manière harmonique pour laquelle les tâches sont ordonnancées via la priorité fixe. L'augmentation de la charge du système via les overheads a donc comme conséquence de décaler les taux de succès d'une VM (le taux de succès à deux VMs avec overhead ressemble fortement à celui à une seule VM sans overhead).

Les résultats avec plus de VMs ne seront pas présentés, ceux-ci n'étant pas pertinents et trop faibles. Cela s'explique principalement par une charge du système bien plus élevée du fait de l'overhead et un système trop chargé pour autant de VMs avec overheads.

Au final, le comportement de cet algorithme est attendu : il ne fonctionne dans aucun des cas pour un système entier constitué de quatre VMs et fonctionne la plupart du temps pour deux ou trois VMs. Les cas où trois VMs sont ordonnancées de manière harmonique décroît avec l'utilisation, ce qui indique qu'avec une utilisation faible, ce cas est le plus courant (jusqu'à 75% des cas). Ainsi, plus l'utilisation est faible, et moins les algorithmes suivants auront de possibilités d'ordonnancement, puisque déjà trois des quatre VMs sont ordonnancées. En revanche, plus l'utilisation croît, et plus le nombre de VMs à ordonnancer sera proche de 2. C'est d'après ces résultats que nous avons, par la suite, cherché un autre algorithme d'ordonnancement pour les VMs, qui vise à compléter les résultats de cet algorithme, pour ordonnancer les VMs restantes qui n'ont pas été prises en compte par celui-ci.

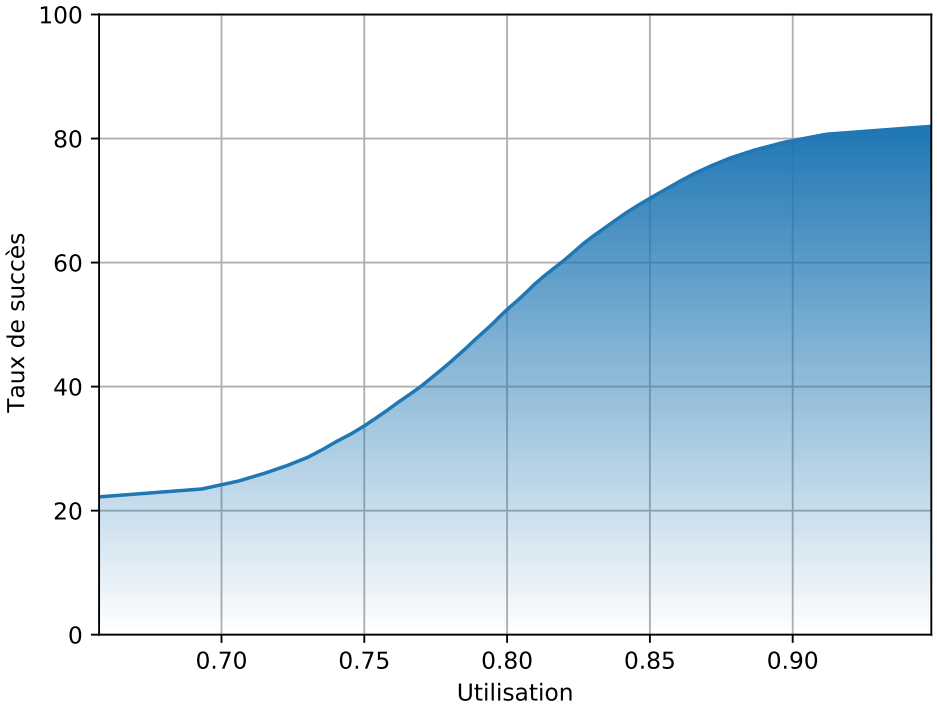


(A) Taux de succès d'exactement une VM en fonction de l'utilisation sans overhead

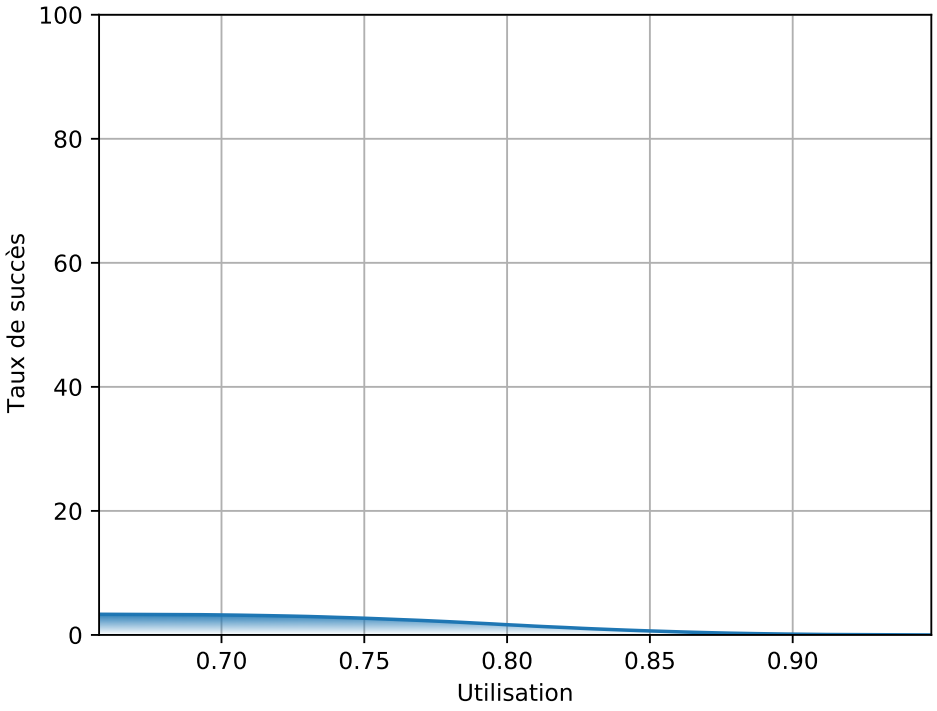


(B) Taux de succès d'exactement une VM en fonction de l'utilisation avec overhead

FIGURE 4.5 : Comparaison entre le taux de succès d'exactement une VM sans et avec overhead



(A) Taux de succès d'exactement deux VMs en fonction de l'utilisation sans overhead



(B) Taux de succès d'exactement deux VMs en fonction de l'utilisation avec overhead

FIGURE 4.6 : Comparaison entre le taux de succès d'exactement deux VMs sans et avec overhead

VM	Tâche τ_i	C_i	T_i	D_i
ρ^1	τ_1	1	20	15
ρ^2	τ_2	1	15	20
ρ^2	τ_3	2	15	30
ρ^3	τ_4	2	20	40
ρ^3	τ_5	1	20	40
ρ^3	τ_6	1	40	40

TABLE 4.3 : Système entier de tâches

4.5 Exemple d'un système où les paramètres des VMs sont inconnus

Dans cette section, nous étudions un exemple de système pour lequel seules les tâches ont des paramètres connus. Nous déduisons ainsi les paramètres dans un premier temps de la première VM, puis des autres VMs, si cela est possible. Nous observons ensuite comment ordonnancer les VMs de manière harmonique grâce aux algorithmes précédemment définis (Algorithmes 2 et 3 pour les paramètres des VMs ainsi que les Algorithmes 4 et 5 pour l'ordonnancement).

L'exemple décrit ici est un exemple type représentatif du projet CEOS [136]. Nous prenons donc ici l'exemple d'un drone autonome. Pour que celui-ci puisse fonctionner, des parties essentielles ont été définies. Parmi celles-ci, il s'agit du contrôle-commande, d'une partie communication pour que le drone puisse recevoir des instructions, et de la partie vidéo, puisque ce drone est un drone de surveillance. La dernière partie permet ainsi d'enregistrer, analyser et de compresser les images des caméras embarquées.

Ces trois parties étant faites par des équipes différentes, chacune travaille sur une VM différente. Il suffit alors de regrouper par la suite leurs VMs sur le drone pour que celui-ci puisse fonctionner. Le problème étant que seules les tâches au sein de ces VMs sont connues. Un dernier travail est alors à fournir : «quand ordonnancer les VMs et pour quelle durée, pour que les tâches des VMs puissent respecter leurs échéances ?».

Nous définissons ainsi les différentes tâches appartenant aux différentes VMs comme dans la Table 4.3. Nous avons ainsi trois jeux de tâches différents, pour chaque VM. Respectivement, $\tau_1(1, 20, 15)$ pour ρ^1, $\tau_2(1, 15, 20)$ et $\tau_3(2, 15, 30)$ pour ρ^2 et enfin $\tau_4(1, 20, 40)$, $\tau_5(1, 20, 40)$ et $\tau_6(1, 40, 40)$ pour ρ^3. L'utilisation des VMs est ainsi fixée à $U^1 = \frac{1}{20} = 0.05$ pour ρ^1, $U^2 = \frac{1}{15} + \frac{2}{15} = 0.2$ pour ρ^2 et $U^3 = \frac{2}{20} + \frac{1}{20} + \frac{1}{40} = 0.175$ pour ρ^3.

Nous fixons également l'overhead des VMs à 1 et le facteur α à 0.1 pour avoir un exemple complet.

L'Algorithme 2 utilisant comme entrée les bornes minimales et maximales sur la période, nous devons alors commencer par définir ces bornes. Ces bornes sont ainsi définies dans les Équations (3.25) et (3.31). Commençons par l'Équation (3.25). Pour obtenir un résultat, nous devons d'abord calculer les résultats des Équations (3.22), (3.23) et (3.24).

$$
\begin{aligned}
T^1_{\beta^1} &\geq \frac{O^1}{1-U^1} \\
&\geq \frac{1}{1-0.05} \\
&\geq 1.05 \\
T^1_{\beta^2} &\geq \frac{O^1}{\alpha} \\
&\geq \frac{1}{0.1} \\
&\geq 10 \\
T^1_{\beta^3} &\geq k \\
&\geq 3 \\
T^1_{min} &\geq \max(\left\lceil T^s_{\beta^1} \right\rceil, \left\lceil T^s_{\beta^2} \right\rceil, T^s_{\beta^3}) \\
&\geq \max(2, 10, 4) \\
&\geq 10
\end{aligned}
$$

La borne minimale est ainsi de 10 d'après les équations précédentes. Nous pouvons maintenant calculer la borne maximale, qui dépend du jeu de tâches associé à la VM. Pour le cas de la VM ρ^1, une seule tâche fait partie du jeu de tâches. Le calcul est donc simple. Reprenons l'Équation (3.31) :

$$
\begin{aligned}
&\forall \tau_i \in \tau, \\
T^1_{max} &\leq \left\lfloor \frac{D_i - C_i}{1-U^1} \right\rfloor \\
&\leq \left\lfloor \frac{15-1}{1-0.05} \right\rfloor \\
&\leq 14
\end{aligned}
$$

La borne maximale sur la période est ainsi de 14. Maintenant que nous avons calculé les deux bornes minimales et maximales sur la période de la VM ρ^1, nous pouvons les utiliser en entrée de l'Algorithme 2.

Algorithme 6 : Résultat de l'Algorithme 2 avec entrées fixées

Entrées : $(T^1_{min} = 10, T^1_{max} = 14, \tau : \tau_1(1, 20, 15))$
Sorties : C^1, T^1, sched

1 $T^1 = 10$;
2 sched = $faux$;
3 $C^1 = \lceil 0.05 \times 10 + 1 \rceil = 2$;
4 $r_1 = 10$ d'après (3.7), $10 \leq 15$, donc τ est ordonnançable;
5 sched = vrai;
6 **retourner** $C^1 = 2$, $T^1 = 10$, *sched = vrai*

Nous définissons le résultat de l'Algorithme 2 dans l'Algorithme 6. Nous décrivons dans ce nouvel algorithme toutes les étapes et les résultats lorsque nous donnons comme entrées à l'Algorithme 2 les valeurs suivantes : $(T^1_{min} = 10, T^1_{max} = 14,$

$\tau : \tau_1(1, 20, 15)$. Nous obtenons ainsi une VM $\rho^1(2, 10, 1)$.

Nous pouvons maintenant déduire les paramètres des autres VMs si ceux-ci peuvent avoir des périodes harmoniques. Cela est fait grâce à l'Algorithme 3. Pour cet algorithme, nous pouvons pré-calculer les T^s_{max} :

$$
\begin{aligned}
&\forall \tau_i \in \tau, \\
T^2_{max} &\leq \left\lfloor \frac{20-1}{1-0.2} \right\rfloor \\
&\leq 23 \\
&\leq \left\lfloor \frac{30-2}{1-0.2} \right\rfloor \\
&\leq 35 \\
T^2_{max} &\leq 23
\end{aligned}
$$

$$
\begin{aligned}
&\forall \tau_i \in \tau, \\
T^3_{max} &\leq \left\lfloor \frac{40-2}{1-0.175} \right\rfloor \\
&\leq 46 \\
&\leq \left\lfloor \frac{40-1}{1-0.175} \right\rfloor \\
&\leq 47 \\
&\leq \left\lfloor \frac{40-1}{1-0.175} \right\rfloor \\
&\leq 47 \\
T^3_{max} &\leq 46
\end{aligned}
$$

Nous décrivons dans l'Algorithme 7, le cheminement pour trouver les paramètres des VMs d'après l'Algorithme 3. Nous obtenons ainsi les VMs $\rho^1(2, 10, 1)$, $\rho^2(5, 20, 1)$ et $\rho^3(8, 40, 1)$.

Maintenant que tous les paramètres de toutes les VMs ont été trouvés, nous pouvons passer à l'ordonnancement de ces VMs. Comme décrit précédemment, l'ordonnancement est donné grâce aux Algorithmes 4 et 5.

Nous présentons dans l'Algorithme 8 les résultats des Algorithmes 4 et 5. Parmi ces résultats, nous pouvons mettre en évidence les ordonnancements trouvés pour les différentes VMs. Pour ρ^1, il s'agit de quatre exécutions différentes : de l'instant 0 à 2, de l'instant 10 à 12, de l'instant 20 à 22 et enfin de l'instant 30 à 32. La VM ρ^2 n'a que deux exécutions : de l'instant 2 à 7 et de l'instant 22 à 27. La VM ρ^3 n'a elle, en revanche, qu'une seule exécution de l'instant 12 à 20.

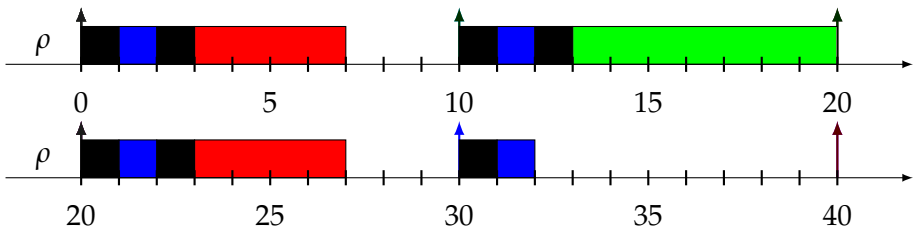


FIGURE 4.7 : Ordonnancements de trois VMs $\rho^1(2, 10, 1)$, $\rho^2(5, 20, 1)$ et $\rho^3(8, 40, 1)$

Algorithme 7 : Résultats de l'Algorithme 3 avec comme entrée l'ensemble de VMs ρ et leur jeu de tâches associé et les paramètres de la VM $\rho^1(2, 10, 1)$

Entrées : ρ
Sorties : listeVM

listeVM = liste();
indiceVM = 1;
début
    $T^2_{max} = 23$;
    $T^2_{test} = T^1 \times 2 = 20$;
    $T^2_{test} = 20 \leq T^2_{max} = 23$, **donc**;
    $C^2_{test} = \lceil 0.2 \times 20 + 1 \rceil = 5$;
    $C^2_{test} = 5 \leq T^1 - C^1 = 8$, **et** $r_2 = 17$ **et** $r_3 = 28$ d'après (3.7) **et** $17 \leq 20$ **et** $28 \leq 30$, **donc** τ est ordonnançable;
    $C^2 = 5$;
    $T^2 = 20$;
    $listeVM+ = \rho^2(5, 20)$;
    indiceVM = 2;
début
    $T^3_{max} = 46$;
    $T^3_{test} = T^2 \times 2 = 40$;
    $T^3_{test} = 40 \leq T^3_{max} = 46$, **donc**;
    $C^3_{test} = \lceil 0.175 \times 40 + 1 \rceil = 8$;
    $C^3_{test} = 8 \leq T^1 - C^1 = 8$, **et** $r_4 = 35$ **et** $r_5 = 38$ **et** $r_6 = 40$ d'après (3.7) **et** $35 \leq 40$ **et** $38 \leq 40$ **et** $40 \leq 40$, **donc** τ est ordonnançable;
    $C^3 = 8$;
    $T^3 = 40$;
    $listeVM+ = \rho^3(8, 40)$;
    indiceVM = 3;
retourner *listeVM*

Algorithme 8 : Résultats des Algorithmes 4 et 5 avec comme entrée $\rho^1(2,10,1)$ et $\rho_{sub} : \rho^2(5,20,1)$ et $\rho^3(8,40,1)$

Entrées : $\rho^1(2,10,1)$, $\rho_{sub} : \rho^2(5,20,1)$ et $\rho^3(8,40,1)$
Sorties : ordoVMs

```
1  ordoVMs = liste();
2  n = 2 + 1;
3  repetitionsVM1 = 40/10 = 4;
4  libre[4] = vrai;
5  ordoVM1 = liste();
6  début
7      débutVM1 = 0 × 10 = 0;
8      finVM1 = 0 × 10 + 2 = 2;
9      ordoVM1 += (0, 2);
10 début
11     débutVM1 = 1 × 10 = 10;
12     finVM1 = 1 × 10 + 2 = 12;
13     ordoVM1 += (10, 12);
14 début
15     débutVM1 = 2 × 10 = 20;
16     finVM1 = 2 × 10 + 2 = 22;
17     ordoVM1 += (20, 22);
18 début
19     débutVM1 = 3 × 10 = 30;
20     finVM1 = 3 × 10 + 2 = 32;
21     ordoVM1 += (30, 32);
22 ordoVMs += ordoVM1;
23 début
24     repetitionsVM = 40/20 = 2;
25     ordoVM = liste();
26     début
27         indice = 0×20/10 = 0;
28         libre[0] = vrai, donc;
29         premierLibre = 0;
30         libre[0] = faux;
31         débutVM = 0 × 10 + 2 = 2;
32         finVM = 0 × 10 + 2 + 5 = 7;
33         ordoVM += (2, 7);
34     début
35         indice = 1×20/10 = 2;
36         libre[2] = vrai, donc;
37         premierLibre = 2;
38         libre[2] = faux;
39         débutVM = 2 × 10 + 2 = 22;
40         finVM = 2 × 10 + 2 + 5 = 27;
41         ordoVM += (22, 27);
42     ordoVMs += ordoVM;
43     début
44         repetitionVM = 40/40 = 1;
45         ordoVM = liste();
46         indice = 0×40/10 = 0;
47         libre[0] = faux, donc test suivant;
48         libre[1] = vrai, donc;
49         premierLibre = 1;
50         débutVM = 1 × 10 + 2 = 12;
51         finVM = 1 × 10 + 2 + 8 = 20;
52         ordoVM += (12, 20);
53     ordoVMs += ordoVM;
54 retourner ordoVMs
```

Nous présentons le résultat final de l'ordonnancement des VMs dans la Figure 4.7. Ici seul l'ordonnancement des VMs est donné (et de leur overhead), et non celui des tâches au sein de chaque VM. L'ordonnancement des tâches au sein de chaque VM étant déjà vérifié par les Équations (3.7) du Chapitre 3, toutes les échéances sont respectées.

Chapitre 5

Ordonnancement de VMs hypervisées non harmoniques : comment ordonnancer les VMs non harmoniques ?

Nous avons vu dans le Chapitre 4 comment trouver les paramètres des VMs pour que les périodes soient harmoniques, et ce afin de pouvoir les ordonnancer à l'aide d'un algorithme simple et efficace. Nous avons également vu des conditions nécessaires sur les périodes et des contraintes sur le temps d'exécution des VMs qui permettent de déterminer des bornes sur les périodes des VMs. Dans certains cas, ces contraintes peuvent être trop importantes et empêcher l'ordonnancement d'un trop grand nombre de jeux de tâches.

Dans ce chapitre, nous cherchons à répondre à cette problématique dans le but d'ordonnancer un maximum de VMs sans les contraintes liées à l'harmonicité des périodes. Nous cherchons ainsi à augmenter le nombre de VMs qui pourront être ordonnancées. Pour ce faire, nous commençons d'abord par présenter avec plus de détails l'approche P-fair. Cette approche a d'abord été développée dans un contexte multi-processeurs, mais s'applique également dans un contexte mono-processeur comme nous le verrons. Une première implémentation de P-fair est ainsi proposée, mais nous présentons ensuite notre approche pour une implémentation de P-fair itérative. Cette approche nous permet ainsi de nous approcher le plus possible d'un ordonnancement P-fair dans un contexte mono-processeur où des VMs peuvent être déjà ordonnancées. Nous améliorons ensuite le temps de calcul de l'implémentation de P-fair itérative pour finir par présenter comment combiner les approches harmoniques et notre approche de P-fair. Le choix s'est ici porté sur P-fair et des algorithmes s'en rapprochant pour une raison principale : plus la VM est ordonnancée souvent et plus les tâches associées à chaque VM respectent leurs échéances. Cela a dans un premier temps été vu grâce à des simulations, mais également d'après la formule de temps de réponse des tâches ordonnancées à l'intérieur de VMs. En effet, moins la VM est exécutée et moins les tâches de la VM pourront être exécutées. Ainsi, chaque intervalle où la VM n'est pas exécutée, est automatiquement ajouté au temps de réponse. Donc plus la VM est exécutée et plus les temps de non exécution sont courts, plus le temps de réponse des tâches sera court.

Sommaire

5.1 Ordonnancement proportionnel (P-fair)

Dans cette section, nous présentons l'algorithme d'ordonnancement P-fair dont la particularité est de répartir équitablement les ressources processeur entre les instances actives des tâches à ordonnancer. Nous nous attachons plus particulièrement à vérifier qu'un ordonnancement donné respecte des contraintes spécifiques pour se rapprocher au mieux d'un ordonnancement dit P-fair. Ces contraintes nous permettent également de quantifier si l'ordonnancement donné est plus ou moins proche d'un ordonnancement P-fair. Nous présentons par la suite comment créer un ordonnancement P-fair à partir de ces contraintes avant d'en voir les limites qui nous ont conduit à créer de nouveaux algorithmes d'ordonnancement se rapprochant de P-fair.

Nous avons déjà présenté l'algorithme d'ordonnancement P-fair dans la Section 2.1. Comme introduit dans [23], cet algorithme a la priorité de produire un ordonnancement équitable et d'être optimal sur une plate-forme multi-processeurs. Il cherche à répartir le plus équitablement possible le temps de calcul entre les différentes tâches. Pour arriver à cela, il utilise un modèle d'ordonnancement fluide idéal qu'il faut ensuite adapter dans un modèle en temps discret. Le modèle fluide idéal est un modèle dans lequel les tâches peuvent être ordonnancées pour une durée arbitrairement petite. Il sert alors de référence à certains algorithmes d'ordonnancement en temps discret qui vont chercher à s'approcher au mieux de l'ordonnancement fluide idéal.

L'ordonnancement P-fair est défini pour un ensemble de tâches τ. L'algorithme d'ordonnancement dans le modèle fluide ne prend en compte que l'utilisation des tâches pour prendre ses décisions d'ordonnancement. Cette propriété se retrouve également dans les algorithmes cherchant à s'approcher de P-fair. L'algorithme d'ordonnancement P-fair a pour contrainte que τ ait une utilisation totale U égale à 1. Si celle-ci est inférieure à 1, la première étape est alors de compléter l'utilisation à l'aide d'une tâche fictive. Cette tâche τ_{fic} aura alors une utilisation égale à $U_{fic} = 1 - U$. Les temps d'exécution de cette tâche représentent alors les temps d'inactivité du processeur, à savoir quand aucune des tâches appartenant initialement à τ n'est exécutée. Grâce à τ_{fic}, il est possible d'utiliser l'algorithme d'ordonnancement P-fair avec n'importe quel jeu de tâches dont l'utilisation totale est inférieure ou égale à 1. Une fois que cette contrainte est respectée, il devient possible de produire un ordonnancement fluide idéal pour τ.

Pour cela, il faut déterminer le pourcentage de temps d'exécution de chaque tâche pour chaque unité de temps, également appelée quantum de temps. Ainsi, durant chaque quantum de temps, toutes les tâches de τ sont exécutées pour une fraction du quantum. C'est à cause de cette particularité que l'ordonnancement P-fair n'est pas directement implantable sur un système réel. En effet, une taille de quantum arbitrairement petite conduit à un nombre de changements de contexte arbitrairement grand. Il faut alors adapter l'ordonnancement fluide pour qu'une seule tâche ne soit exécutée pendant un quantum de temps, et non l'ensemble de toutes les tâches.

C'est pour pallier ce problème que nous allons proposer plusieurs approches pour définir des ordonnancements s'approchant de l'ordonnancement fluide idéal. Mais avant cela, nous allons commencer par définir ce qui fait qu'un ordonnancement est P-fair ou non. Tout d'abord, il a été prouvé dans [23] que pour un système où la charge processeur est inférieure ou égale à 100%, il est possible de trouver un ordonnancement P-fair. Ainsi, dans notre cas, il sera toujours possible de trouver un ordonnancement s'en approchant le plus possible ou répondant totalement aux

critères faisant qu'un ordonnancement est P-fair. Ces critères définis par rapport à l'ordonnancement fluide idéal seront détaillés par la suite.

Un ordonnancement est dit P-fair s'il respecte les conditions que nous allons maintenant rappeler. En effet, ces conditions ont été explicitées et prouvées dans [23]. La première de ces conditions s'applique au jeu de tâches τ et est définie comme suit :

$$\sum_{\tau_i \in \tau} U_i \leq 1$$

Cette condition, initialement définie comme $\sum_{\tau_i \in \tau} U_i \leq m$ dans un contexte multi-cœurs où m est le nombre de cœurs, a été adaptée au cas d'un seul cœur. La fonction $S(k,t)$ est également définie et vaut 1 si la tâche τ_k a été ordonnancée à l'instant t ou 0 sinon. Cette fonction est utilisée pour définir la notion de lag qui correspond à la différence entre la durée d'exécution dans le modèle fluide de la tâche et la durée d'exécution effective dans l'ordonnancement discret. Le lag de la tâche τ_k à l'instant t, $lag(k,t)$ est défini de la manière suivante :

$$lag(k,t) = U_k \times t - \sum_{l \in [0,t[} S(k,l) \tag{5.1}$$

Nous utilisons le lag pour définir une condition permettant de déterminer si un ordonnancement respecte les contraintes d'un ordonnancement P-fair. Cette condition est définie pour un ensemble de tâches τ par la relation suivante :

$$\forall \tau_i \in \tau, \forall t \in \mathbb{N},$$
$$-1 < lag(i,t) < 1$$

Après avoir présenté comment un algorithme d'ordonnancement fonctionnant en temps discret a été introduit dans [23], nous étudions les limites de cet algorithme. À partir de cette étude, nous proposons de nouveaux algorithmes dont les objectifs sont de produire des ordonnancements qui, dans notre contexte, s'approchent au mieux de l'ordonnancement fluide idéal.

Nous avons d'abord besoin de définitions supplémentaires pour pouvoir utiliser l'algorithme proposé dans [23]. Parmi celles-ci, nous trouvons la chaîne de caractères caractéristique $str(k,t)$ de la tâche τ_k à l'instant t. Cette chaîne peut être constituée des éléments de l'ensemble $\{-, 0, +\}$. Elle se définit ainsi :

$$str(k,t) = signe(U_k \times (t+1) - \lfloor U_k \times t \rfloor - 1)$$

L'algorithme d'ordonnancement P-fair va ainsi se baser sur la chaîne de caractères caractéristique et le lag pour décider quelle tâche exécuter à l'instant t. Pour résumer les prises de décision, les tâches sont classées d'après leur lag et la chaîne de caractères caractéristique en trois catégories :

- Une tâche τ_i est urgente à l'instant t, si et seulement si, $lag(i,t) > 0$ et $str(i,t) \neq -$.
- Une tâche τ_i est en avance à l'instant t, si et seulement si, $lag(i,t) < 0$ et $str(i,t) \neq +$.
- Une tâche τ_i est en compétition à l'instant t si et seulement si elle n'est ni urgente ni en avance.

À partir de ces trois catégories, il est ensuite possible d'ordonnancer les tâches à chaque instant d'après une suite de règles dont nous donnons des exemples. Par exemple, si la tâche τ_i est en avance au temps t alors cette tâche n'est pas ordonnancée à l'instant t, donc $S(i,t) = 0$. Un autre exemple consiste en un ordonnancement obligatoire de tâche. En effet, si la tâche τ_i est urgente à l'instant t, alors la tâche est ordonnancée à l'instant t, donc $S(i,t) = 1$.

Ainsi, grâce à ces règles (d'autres sont définies dans [23]), il est possible de déterminer quelle tâche exécuter à quel moment. Le principal inconvénient de ces règles est que les surcoûts ne sont pas pris en compte et qu'un grand nombre de préemptions est créé.

5.2 Implémentations de P-fair

Dans cette section, nous commençons par présenter un algorithme pour trouver un ordonnancement qui s'approche le plus du modèle fluide idéal présenté dans [23] à l'aide de ressources de calcul infinies. Il existe déjà différentes implémentations de P-fair, comme dans [4], [9] ou bien [87] mais ces implémentations sont faites pour une architecture multi-processeurs et ne reposent pas sur un ordonnancement itératif, ce qui est fondamental pour notre algorithme. Nous proposons ainsi un autre algorithme d'ordonnancement itératif, qui vise à se rapprocher du modèle fluide idéal. Cet algorithme est plus éloigné de l'ordonnancement P-fair dans un modèle fluide (la mesure se fait grâce au lag) que la méthode précédente mais permet de trouver plus facilement un résultat. Il permet également d'ordonnancer les VMs de manière itérative, à savoir les unes après les autres et dans un contexte mono-processeur, ce qui reste opposé à l'utilisation habituelle de P-fair.

5.2.1 Cas des ressources de calcul infinies

Dans cette première implémentation de P-fair, nous pallions les problèmes de l'ordonnancement P-fair évoqués dans la Section 5.1, à savoir la prise en compte des overheads, et le calcul d'un ordonnancement pour toutes les VMs simultanément. Pour ce faire, nous présentons une solution utilisant l'optimisation linéaire afin de déterminer un ordonnancement le plus proche de P-fair possible.

Définition 49 (Optimisation linéaire)
L'optimisation linéaire est une méthode d'optimisation mathématique qui vise à minimiser une fonction objectif linéaire sujette à des contraintes sous forme d'équations ou inéquations linéaires.

Définition 50 (Fonction linéaire à variables multiples)
Dans sa forme générale, une fonction linéaire avec k variables multiples est une fonction définie par :

$$f(x_1, \ldots, x_k) = b + a_1 \times x_1 + \cdots + a_k \times x_k$$

Où $a_1, \ldots, a_k$ et b sont des constantes.

Ainsi, nous définissons une fonction objectif et différentes équations et inéquations qui servent de contraintes dans le but de trouver un ordonnancement répondant aux contraintes de P-fair. Pour cela, nous utilisons plusieurs notions vues dans la Section 5.1, à savoir, le lag, et la fonction $S(k,t)$. Notre but étant de trouver un ordonnancement des VMs, celui-ci est donc appliqué aux VMs et non aux tâches. Le seul paramètre à définir pour cet algorithme est soit l'utilisation des tâches (ou un

majorant) ou soit l'utilisation maximale souhaitée pour les VMs, il sera alors possible de l'utiliser dans les deux contextes.

Le problème d'optimisation linéaire, pour trouver un ordonnancement à l'ensemble des VMs ρ de l'instant 0 à l'instant $H(\rho)$ où $H(\rho)$ est l'hyper-période des VMs calculée à partir de leur utilisation, se définit ainsi :

$$minimiser \sum_{k,t} (|lag(k,t)|) \tag{5.2}$$

$$\forall t, \sum_{k} S_{k,t} \leq 1 \tag{5.3}$$

$$\forall k, lag(k,0) = 0 \tag{5.4}$$

$$S_{k,t} = \begin{cases} 1, & \text{si } lag(k,t) = \max\limits_{\rho^l \in \rho} (lag(l,t)) \\ 0, & \text{sinon.} \end{cases} \tag{5.5}$$

$$lag(k,t) = U^k \times t - \sum_{l \in [0,t[} S_{k,l} \tag{5.6}$$

$$\forall k, \frac{\sum\limits_{t} S_{k,t}}{H(\rho)} = U^k \tag{5.7}$$

Définition 51 (Hyper-période de VMs)
Lorsque les VMs sont activées de manière synchrone, l'hyper-période des VMs est la période sur laquelle leur schéma d'exécution se répète. Cette hyper-période est calculée à partir des utilisations des VMs sous forme fractionnaire. L'utilisation d'une VM étant égale au temps d'exécution de la VM sur sa période, il suffit alors de calculer le plus petit commun multiple des numérateurs des utilisations des VMs pour trouver l'hyper-période.

Le problème d'optimisation linéaire se décompose ainsi : la fonction objectif est définie dans l'Équation (5.2). Il s'agit de minimiser la somme de la valeur absolue des lags de chaque VM à chaque instant. En effet, le lag est défini comme l'écart entre l'ordonnancement dans un modèle fluide et l'ordonnancement actuel. Celui-ci pouvant être négatif ou positif selon l'avance ou le retard des tâches par rapport à l'ordonnancement du modèle fluide. Notre but est alors de nous approcher le plus possible d'un lag de 0, à chaque instant et pour chaque VM. Cela est dû à la définition du lag (5.1) : plus le lag est proche de 0, et plus l'ordonnancement est proche de P-fair.

La première contrainte, Équation (5.3), permet de vérifier à chaque instant que le nombre de VMs pouvant être exécutées ne dépasse pas 1. Cette contrainte est due au fait que nous ne prenons en compte qu'un seul cœur. Cette contrainte est à changer dans un cas multi-cœurs pour correspondre au nombre de cœurs utilisables.

La seconde contrainte, Équation (5.4) permet d'initialiser le lag des différentes VMs à l'instant 0. En effet, celles-ci n'ayant pas encore été exécutées, leur lag de départ est donc de 0. Cette initialisation est nécessaire pour que la résolution du problème d'optimisation linéaire puisse trouver le meilleur ordonnancement possible. Elle est due à la définition même du lag, qui, lorsqu'aucune tâche n'a été ordonnancée, est forcément égal à 0.

La troisième contrainte, Équation (5.5) permet quant à elle de définir quelle VM ordonnancer à chaque instant. En effet, $S_{k,t}$ est une fonction qui permet de définir si oui ($S_{k,t} = 1$) ou non ($S_{k,t} = 0$) la VM est ordonnancée à l'instant t. Pour ce faire, cette contrainte ne sélectionne qu'une seule VM : la VM dont le lag est le plus grand. Ainsi, la VM avec le plus de retard est sélectionnée pour être exécutée. Cette contrainte est

également à modifier pour l'appliquer dans un contexte multi-cœurs : plusieurs VMs devront alors être sélectionnées.

La quatrième contrainte, Équation (5.6) permet de définir le lag de chaque VM à chaque instant. C'est donc le même calcul que présenté dans [23], mais appliqué au contexte du problème d'optimisation linéaire.

La dernière contrainte, Équation (5.7) permet de s'assurer que l'utilisation de chaque VM correspond bien à l'ordonnancement calculé. Pour chaque VM, cette équation vérifie donc combien d'unités de temps elle a été exécutée et divise ce temps par l'hyper-période des VMs pour vérifier que le résultat est bien égal à l'utilisation de la VM.

Le problème d'optimisation linéaire précédemment défini souffre de deux problèmes : les overheads liés aux VMs ne sont pas pris en compte, et le calcul se fait pour toutes les VMs en même temps, il n'est donc pas possible d'ajouter, par exemple, des VMs à un ordonnancement de VMs donné. Or, nous avions déjà défini un algorithme pour ordonnancer le plus de VMs possibles avec des périodes harmoniques (Chapitre 4), notre but est donc ici de pouvoir ajouter de nouvelles VMs lorsqu'un tel ordonnancement a déjà été trouvé.

Pour pallier ces problèmes, nous proposons ainsi deux solutions. La première concerne le problème de prise en compte des overheads. Cette solution consiste à ajouter une nouvelle contrainte permettant de fixer un temps O_{min} d'exécution minimal à chaque fois que la VM est exécutée. La contrainte est alors :

$$\forall \rho^k \in \rho, t \in [0, H(\rho) - O_{min}[\text{ , si } S_{k,t} = 1, \text{ alors } \forall t' \in [1, O_{min}], S_{k,t+t'} = 1 \tag{5.8}$$

Pour pallier le second problème, à savoir trouver un ordonnancement le plus proche de P-fair tenant compte des VMs déjà attribuées, nous proposons de remplacer la contrainte de l'Équation (5.5) par deux autres contraintes. Une première contrainte se fait sur l'ensemble des VMs déjà ordonnancées ρ_{harmo} et l'autre sur l'ensemble des autres VMs à ordonnancer ρ_{pfair} :

$$\forall \rho^k \in \rho_{harmo}, \forall t \in [0, H(\rho)[\text{ ,}$$
$$S_{k,t} = \begin{cases} 1, & \text{si } \rho^k \text{ est ordonnancée à l'instant } t \\ 0, & \text{sinon.} \end{cases} \tag{5.9}$$
$$\forall \rho^k \in \rho_{pfair}, \forall t \in [0, H(\rho)[\text{ ,}$$
$$S_{k,t} = \begin{cases} 1, & \text{si } L_{k,t} = \max\limits_{\rho^l \in \rho} (L_{l,t}) \\ 0, & \text{sinon.} \end{cases} \tag{5.10}$$

La première contrainte, Équation (5.9) est donc appliquée à l'ensemble des VMs ρ_{harmo} déjà ordonnancées. Cette équation permet de définir, pour chaque VM à chaque instant, si elle est ordonnancée ou non. Ces données sont donc déjà calculées et données en entrée au problème d'optimisation linéaire.

La seconde contrainte, Équation (5.10) définit, comme dans l'Équation (5.5), la VM à exécuter dans l'ensemble des VMs qui n'ont pas encore été ordonnancées par rapport au lag.

La dernière étape de cette solution est donc de résoudre le problème d'optimisation linéaire. Pour ce faire, différentes solutions sont possibles, et différents algorithmes sont disponibles. Parmi les solutions pour résoudre un tel problème, il existe

un grand nombre de solveurs utilisant l'algorithme du simplexe ou les méthodes de points intérieurs. Le choix du solveur est ainsi laissé libre selon les besoins.

Ces algorithmes et solutions pour résoudre ce problème ne faisant pas partie du cadre de cette thèse, les solutions ne seront pas plus détaillées. Bien que ce problème d'optimisation linéaire permette de trouver une solution respectant au mieux l'ordonnancement P-fair, nous proposons d'autres solutions qui donnent des résultats plus éloignés d'un ordonnancement P-fair, mais bien plus rapides en temps de calcul. En effet, une telle solution basée sur de l'optimisation linéaire peut demander des temps de calcul élevés pour obtenir une solution et ne pourrait donc être utilisable qu'à l'aide de ressources de calculs très importantes. Pour donner un exemple, le problème linéaire précédemment défini pouvait trouver un résultat pour trois VMs ordonnancées pendant 20 unités de temps en environ trente minutes, mais au delà de 20 unités de temps, aucun résultat n'a été trouvé en moins de 24 heures. Ce point est abordé plus en détails dans le Chapitre 7.

5.2.2 Algorithme proportionnel itératif

Dans cette section, nous proposons un algorithme itératif s'approchant du modèle fluide de P-fair avec un temps de calcul raisonnable. Ce point s'oppose au problème d'optimisation linéaire présenté dans la Section 5.2.1. En effet, la méthode précédente avait un temps de calcul exponentiel mais proposait un ordonnancement s'approchant le plus possible du modèle fluide. De plus, ce nouvel algorithme prend en compte les VMs (ou tâches) déjà ordonnancées. Nous pouvons ainsi l'utiliser de manière itérative : l'ordonnancement de chaque VM est trouvé à tour de rôle en prenant en compte les VMs déjà ordonnancées. Cela aura donc pour avantage de pouvoir être utilisé après l'algorithme d'ordonnancement harmonique présenté dans le Chapitre 4.

Pour résumer l'algorithme présenté dans la suite de cette section, il s'agit d'abord de définir un ordonnancement aléatoire pour la VM dont nous déterminons l'ordonnancement, puis de tester son lag. À partir de son lag, à chaque instant, nous pouvons sélectionner, au moyen d'un déplacement d'instant d'exécution, un lag approchant l'ordonnancement de la VM du modèle fluide P-fair.

Définition 52 (Déplacement d'instant d'exécution)
Un déplacement d'instant d'exécution pour une VM ρ^k à l'instant t consiste à déplacer l'exécution de la VM d'un instant donné t à un autre instant t' où la VM n'était pas exécutée. Ici t' peut être plus grand ou plus petit que t.

Définition 53 (Déplacement d'instant d'exécution avant)
Un déplacement avant consiste à déplacer l'exécution de la VM d'un instant t à un autre instant t' où la VM n'était pas exécutée et où $t' < t$.

Définition 54 (Déplacement d'instant d'exécution arrière)
Un déplacement arrière consiste à déplacer l'exécution de la VM d'un instant t à un autre instant t' où la VM n'était pas exécutée et où $t < t'$.

Nous proposons dans la Table 5.1 un exemple pour mettre en avant les déplacements d'instants d'exécution. La table se présente ainsi : la première colonne représente le temps, puis la colonne suivante indique par la variable S si la VM ρ^k est exécutée ou non à l'instant t (égale à 0 si la VM n'est pas exécutée, 1 sinon). La colonne suivante indique, quant à elle, le lag à l'instant t de la VM ρ^k. Les deux colonnes suivantes représentent ensuite l'exécution de la VM indiquée par la variable S et son lag après avoir effectué un déplacement d'instant d'exécution avant, et les

	Référence			Déplacement avant			Déplacement arrière	
$t = 0$	S = 0	lag = 0		S = 1	lag = 0		S = 0	lag = 0
$t = 1$	S = 1	lag = 0.6		S = 1	lag = -0.4		S = 0	lag = 0.6
$t = 2$	S = 1	lag = 0.2		S = 1	lag = -0.8		S = 1	lag = 1.2
$t = 3$	S = 1	lag = -0.2		S = 0	lag = -1.2		S = 1	lag = 0.8
$t = 4$	S = 0	lag = -0.6		S = 0	lag = -0.6		S = 1	lag = 0.4
$t = 5$	S = 0	lag = 0		S = 0	lag = 0		S = 0	lag = 0

TABLE 5.1 : Exemples de déplacements d'instants d'exécution avant et arrière avec un ordonnancement de référence pour une VM ρ^k où $U^k = 0.6$

deux colonnes suivantes représentent l'exécution de la VM indiquée par la variable S et son lag après avoir effectué un déplacement d'instant d'exécution arrière.

La VM de référence ρ^k ici présentée a une utilisation U^k de 0.6. La VM n'est exécutée qu'à trois instants dans l'ordonnancement de référence : 1, 2 et 3. Lors du déplacement d'instant d'exécution avant, l'exécution de la VM à l'instant 3 passe à l'instant 0 tandis que lors du déplacement arrière, l'exécution de la VM à l'instant 1 passe à l'instant 4.

Lors du déplacement d'instant d'exécution avant, nous mettons en avant les trois lags qui changent : à savoir à l'instant 1, 2 et 3. Nous notons ici que les lags sont décrémentés de 1. Lors du déplacement arrière, les trois lags qui changent sont quant à eux incrémentés de 1.

Avant d'étudier les différentes propriétés des déplacements d'instants d'exécution, nous montrons que le calcul du lag peut s'exprimer de manière itérative. Cette formulation est utile pour déterminer les conséquences d'un déplacement d'instant d'exécution et permet d'éviter une redondance de calculs à chaque instant, ce qui sera utile pour les prochains algorithmes. Nous proposons ainsi une nouvelle formule pour le lag, qui dépend du lag précédent :

$$lag(k,t) = lag(k,t-1) + U^k - S(k,t-1) \tag{5.11}$$

Où $lag(k,0) = 0$.

Démonstration. Nous démontrons par récurrence l'Équation (5.11). Commençons par montrer que l'équation est vraie pour $t = 1$. Nous avons $lag(k,0) = 0$ et :

$$lag(k,1) = U^k \times 1 - \sum_{l \in [0,1[} S(k,l) = U^k - S(k,0) = lag(k,0) + U^k - S(k,0)$$

L'équation (5.11) est donc vraie pour $t = 1$. Supposons que l'équation l'Équation (5.11) soit vraie à l'instant t :

$$lag(k,t) = lag(k,t-1) + U^k - S(k,t-1) \tag{5.12}$$

Démontrons alors que cette égalité reste vraie au rang $t+1$. Commençons par déterminer l'Équation (5.1) au rang $t+1$:

$$lag(k,t+1) = U_k \times (t+1) - \sum_{l \in [0,t+1[} S(k,l) \tag{5.13}$$

Nous avons alors au rang $t+1$:

$$\begin{aligned} lag(k,t+1) &= U_k \times (t+1) - \sum_{l \in [0,t+1[} S(k,l) \\ &= U^k \times t - \sum_{l \in [0,t[} S(k,l) + U^k - S(k,t) \\ &= lag(k,t) + U^k - S(k,t) \end{aligned}$$

La récurrence est ainsi vérifiée au rang $t+1$, l'hypothèse de récurrence est donc vraie. □

Théorème 4 (Conséquences d'un déplacement d'instant d'exécution avant)
Prenons un ordonnancement donné avec les lags pour la VM ρ^k définis à chaque instant. Un déplacement d'instant d'exécution avant pour la VM ρ^k de l'instant t à l'instant t' où $t' < t$ décrémente le lag de 1 dans l'intervalle $]t', t]$.

Démonstration. Pour démontrer le Théorème 4, nous reprenons les lags itératifs de l'instant t' à $t+1$:

$$\begin{aligned} lag(k,t') &= lag(k,t'-1) + U^k - S(k,t'-1) \\ lag(k,t'+1) &= lag(k,t') + U^k - S(k,t') \\ lag(k,t) &= lag(k,t-1) + U^k - S(k,t-1) \\ lag(k,t+1) &= lag(k,t) + U^k - S(k,t) \end{aligned}$$

Nous avons donc deux informations : $S(k,t') = 0$ et $S(k,t) = 1$, puisqu'il s'agit d'avancer l'exécution de la VM ρ^k de l'instant t à t'. Nous obtenons alors $S(k,t') = 1$ et $S(k,t) = 0$. Ainsi, l'inversion implique que $lag(k,t'+1)$ est diminué de 1, ainsi que tous les autres lags jusqu'à l'instant $t+1$, d'après l'Équation (5.11). □

Théorème 5 (Conséquences d'un déplacement d'instant d'exécution arrière)
Prenons un ordonnancement donné avec les lags pour la VM ρ^k définis à chaque instant. Les conséquences d'un déplacement d'instant d'exécution arrière de l'instant t à l'instant t' où $t < t'$ incrémente le lag de 1 dans l'intervalle $]t, t']$.

Démonstration. Pour démontrer le Théorème 5, reprenons les lags itératifs de l'instant t à $t'+1$:

$$\begin{aligned} lag(k,t) &= lag(k,t-1) + U^k - S(k,t-1) \\ lag(k,t+1) &= lag(k,t) + U^k - S(k,t) \\ lag(k,t') &= lag(k,t'-1) + U^k - S(k,t'-1) \\ lag(k,t'+1) &= lag(k,t') + U^k - S(k,t') \end{aligned}$$

Sur cet ordonnancement nous avons donc deux informations : $S(k,t') = 1$ et $S(k,t) = 0$, puisqu'il s'agit de retarder l'ordonnancement de la VM ρ^k de l'instant t à t'. L'ordonnancement devient alors $S(k,t') = 0$ et $S(k,t) = 1$. Ainsi, l'inversion implique que $lag(k,t+1)$ est augmenté de 1, ainsi que tous les autres lags jusqu'à l'instant $t'+1$, d'après l'Équation (5.11). □

Ces deux Théorèmes (4 et 5) nous permettent ainsi de déterminer l'impact sur le lag des différents déplacements. Avant de pouvoir utiliser un algorithme pour s'approcher le plus possible d'un ordonnancement P-fair, il nous faut une métrique globale pour comparer deux ordonnancements et ainsi déterminer celui qui sera le plus proche d'un ordonnancement P-fair. Pour cela, nous allons utiliser le même principe que la fonction objectif du problème d'optimisation linéaire (5.2).

En effet, l'ordonnancement le plus proche du modèle fluide peut être évalué par rapport au lag, et plus précisément, grâce à la somme de la valeur absolue des lags à chaque instant. Ainsi, plus cette somme est proche de 0 et plus le lag est proche du modèle fluide.

L'algorithme, pour trouver un ordonnancement P-fair, se base ainsi sur le lag : le but est alors de déterminer, à partir d'un ordonnancement aléatoire donné, le plus grand impact que peut avoir un déplacement d'instant d'exécution (avant ou arrière). Nous cherchons alors à réduire au maximum la somme des valeurs absolues des lags et d'itérer cette opération jusqu'à ne plus avoir d'améliorations possibles. Nous le définissons dans l'Aglorithme 9.

Algorithme 9 : Trouver l'ordonnancement le plus proche de P-fair

Entrées : U^i, booléen[] déjàOrdo, tmax
Sorties : booléen[] ordoVM

```
1 booléen[] base = OrdoAléatoire(U^i, déjàOrdo, tmax);
2 plusFair = AméliorerPfair(base, U^i, déjàOrdo, tmax);
3 tant que base ≠ plusFair faire
4     base = plusFair;
5     plusFair = AméliorerPfair(base, U^i, déjàOrdo, tmax);
6 retourner plusFair
```

Dans cet algorithme, les entrées sont :

- U^i, l'utilisation de la VM ρ^i pour laquelle nous souhaitons trouver un ordonnancement.
- booléen[] déjàOrdo, un tableau de booléens représentant l'ordonnancement des VMs déjà ordonnancées. Cette valeur est ainsi un équivalent de la fonction $S(k,t)$: le booléen est à $vrai$ si une VM est déjà ordonnancée à l'instant t et à $faux$ sinon.
- tmax, le temps maximum sur lequel nous devons calculer l'ordonnancement de la VM ρ^i. Nous verrons plus tard lors d'exemples dans la Section 5.4 comment calculer cette valeur pour qu'elle soit la plus faible possible.

Nous proposons de plus une équation pour déterminer l'utilisation qui sera accordée à chaque VM à ordonnancer. Pour ce faire, nous devons ajouter deux notations différentes : l'une correspond à l'utilisation des tâches d'une VM et l'autre à l'utilisation réelle de la VM. La somme des utilisations des tâches de la VM ρ^k est notée U^k_τ et l'utilisation réelle de la VM accordée par le système est notée U^k. L'ensemble des VMs déjà ordonnancées est noté ρ_o et les VMs non ordonnancées et à ordonnancer sont notées ρ_n

Le calcul se fait ainsi pour chaque VM ρ^k dont l'utilisation U^k accordée par l'algorithme est à déterminer en répartissant l'utilisation restante pour obtenir une utilisation totale pour toutes les VMs de 1 :

$$U^k = \frac{U^k_\tau}{\sum\limits_{\rho^s \in \rho_n} U^s_\tau} \times (1 - \sum_{\rho^s \in \rho_o} U^s) \tag{5.14}$$

Pour trouver le temps maximum sur lequel l'algorithme doit déterminer l'ordonnancement, nous proposons d'évaluer l'hyper-période des VMs d'après la forme fractionnaire irréductible de leur utilisation. En effet, puisque l'utilisation d'une VM correspond au temps d'exécution de la VM divisé par sa période, la forme fractionnaire irréductible permet ainsi d'obtenir une période virtuelle pour les VMs qui ne sont pas encore ordonnancées et pour lesquelles seule l'utilisation est pour l'instant connue. L'hyper-période sera alors le plus petit commun multiple entre les périodes des VMs avec paramètres affectés et celles des VMs non affectées avec leurs périodes virtuelles.

La sortie de l'algorithme est un tableau de booléens de taille *tmax* représentant l'ordonnancement de la VM ρ^i. Ce tableau est utilisé de la même manière que l'entrée déjàOrdo. La première ligne de l'algorithme (Ligne 1) permet de trouver un nouvel ordonnancement aléatoire pour initialiser l'algorithme à l'aide de l'Algorithme 10. La seconde ligne (Ligne 2) permet de trouver un premier ordonnancement plus proche de P-fair à l'aide de l'Algorithme 11.

Les lignes suivantes de l'algorithme permettent de continuer de trouver des ordonnancements plus proches de P-fair jusqu'à ce qu'aucune amélioration ne soit possible. Ainsi, la Ligne 3, est une boucle qui continue jusqu'à ce que les ordonnancements des variables *base* et *plusFair* ne soient plus différentes. La ligne suivante (Ligne 4) actualise la variable *base*, tandis que la Ligne 5 calcule un autre ordonnancement plus proche de Pfair pour la *base* actualisée. Ces améliorations se poursuivent jusqu'à ce qu'aucune ne soit possible pour finir par retourner le résultat de l'ordonnancement trouvé (Ligne 6).

Algorithme 10 : OrdoAléatoire

Entrées : U^i, booléen[] déjàOrdo, tmax
Sorties : booléen[] ordoVM

```
1 nombreAOrdonnancer = U^i × tmax;
2 booléen[] ordoVM = booléen[tmax];
3 pour i = 0 ; i < nombreAOrdonnancer ; i = i + 1 faire
4     a = aléatoire(0, tmax);
5     si déjàOrdo[a] = 0 et ordoVM[a] = 0 alors
6         ordoVM[a] = 1;
7     sinon
8         i = i - 1;
9 retourner ordoVM
```

L'Algorithme 10 est utilisé à la Ligne 1 de l'Algorithme 9. Il permet ainsi de définir un nouvel ordonnancement aléatoire pour la VM ρ^i pour laquelle nous souhaitons trouver un nouvel ordonnancement proche de P-fair. Les entrées de cet algorithme sont les mêmes que pour l'Algorithme 9 et la sortie de l'algorithme est un tableau de booléens représentant le nouvel ordonnancement aléatoire trouvé.

La première ligne de l'algorithme (Ligne 1) définit le nombre d'instants où la VM devra être ordonnancée. Cela correspond donc au temps d'exécution de la VM. La seconde ligne (Ligne 2) initialise, quant à elle, le tableau de booléens représentant l'ordonnancement de la VM. Il est donc de la même taille que *tmax*. La boucle

suivante (Ligne 3) se répète autant de fois qu'il faut exécuter la VM. Ainsi, nous cherchons un instant aléatoire où ordonnancer la VM (Ligne 4). Si aucune VM n'est exécutée à cet instant (Ligne 5), la VM étudiée est alors exécutée à cet instant (Ligne 6). Sinon nous recommençons la boucle (Ligne 8). Une fois tous les instants d'exécution trouvés, nous retournons le résultat (Ligne 9).

Algorithme 11 : AméliorerPfair

```
   Entrées : booléen[] ordoVM, U^i, booléen[] déjàOrdo, tmax
   Sorties : booléen[] ordoVM
 1 impactMax = 0;
 2 début = 0;
 3 fin = 0;
 4 lags = CalculLags(ordoVM, U^i, tmax);
 5 lagsPlus1 = ajouter(lags, 1);
 6 lagsMoins1 = ajouter(lags, -1);
 7 pour i = 0 ; i < tmax ; i = i + 1 faire
 8     si déjàOrdo[i] = 1 alors
 9         continue;
10     pour j = i + 1 ; j < tmax ; j = j + 1 faire
11         si déjàOrdo[j] = 1 et ordoVM[j] != ordoVM[i] alors
12             continue;
13         section1 = lags[i :j];
14         somme = 0;
15         section;
16         si lag[i] == 1 alors
17             section = lagsPlus1[i :j];
18         sinon
19             section = lagsMoins1[i :j];
20         pour k = 0 ; k < j - i ; k = k + 1 faire
21             impact = |section[k]| - |section1[k]|;
22             somme += impact;
23         si somme < impactMax alors
24             impactMax = somme;
25             début = i;
26             fin = j;
27 ordoVM[i] = !ordoVM[i];
28 ordoVM[j] = !ordoVM[j];
29 retourner ordoVM
```

L'Algorithme 11 est appelé aux Lignes 2 et 5 de l'Algorithme 9. Il correspond donc à l'algorithme qui va chercher à améliorer un ordonnancement donné pour le rendre plus proche d'un ordonnancement P-fair. Cette amélioration se fera à l'aide d'un déplacement d'instant d'exécution dans un emplacement libre, que ce soit avant ou arrière.

Les entrées de cet algorithme sont donc un tableau de booléens, appelé *ordoVM* pour représenter l'ordonnancement déjà donné à la VM, donc l'ordonnancement que nous cherchons à améliorer. Les deux autres entrées sont un autre tableau de booléens, *déjàOrdo*, représentant l'ordonnancement des autres VMs déjà ordonnancées

et $tmax$, représentant la taille de l'ordonnancement à trouver. Le tableau *déjàOrdo* est donc entièrement à $faux$ si aucune VM n'a été précédemment ordonnancée. En ce qui concerne la sortie, cet algorithme renvoie un tableau de booléens représentant le nouvel ordonnancement de la VM étudiée (ou le même si aucun changement n'a été trouvé).

Les Lignes 1 à 6 permettent d'initialiser les variables nécessaires à l'algorithme. Parmi elles, $impactMax$ (Ligne 1), permettra au cours de l'algorithme de comparer les différences entre les lags calculés pour trouver celui qui aura le meilleur impact pour se rapprocher au mieux d'un ordonnancement P-fair. Nous avons également les variables *début* (Ligne 2) et fin (Ligne 3) qui permettent de connaître les indices à intervertir pour effectuer le déplacement d'instants d'exécution. Nous avons ensuite trois tableaux différents représentant le lag. Le premier tableau, $lags$ représente le lag de la VM étudiée à chaque instant. Il est calculé à l'aide de l'Algorithme 12. Les deux autres tableaux de lags, $lagsPlus1$ (Ligne 5) et $lagsMoins1$ (Ligne 6) représentent respectivement chaque lag, à chaque instant, incrémenté de 1, et chaque lag à chaque instant décrémenté de 1. Ces deux tableaux sont pré-initialisés pour effectuer moins de calculs au cours de cet algorithme mais également pour des questions de lisibilité. Ces tableaux sont dus aux déplacements d'instants d'exécution (Théorèmes 4 et 5) : nous avons ainsi les conséquences de lags pré-calculées à chaque instant.

Passons à la Ligne 7. Il s'agit d'une première boucle qui va tester tous les indices du tableau $ordoVM$. Cette boucle permettra de déterminer le premier instant t d'un déplacement d'instant d'exécution. La Ligne 8 permet de tester si l'emplacement sélectionné, à savoir l'indice i est un emplacement occupé par les VMs déjà ordonnancées. Si tel est le cas, cette valeur est directement passée (Ligne 9), sinon, nous pouvons la tester et passer aux lignes suivantes.

La seconde boucle, Ligne 10 commence de l'indice $i+1$ jusqu'à $tmax$. Elle permet ainsi de déterminer l'instant t' de fin pour le déplacement d'instant d'exécution. Elle se fera ainsi entre les indices i et j. La première étape de cette boucle est la même que la boucle précédente : tester si à cet instant aucune VM n'est déjà ordonnancée (Ligne 11) et passer l'instant si tel est le cas (Ligne 12). Nous y ajoutons une condition supplémentaire consistant à vérifier s'il est possible d'inverser les exécutions des VMs aux instants i et j et donc s'ils sont différents.

Les lignes suivantes représentent l'algorithme proportionnel itératif à proprement parler. Les Lignes 13 à 14 initialisent des variables pour les calculs. La Ligne 13 extrait le lag du tableau de lags pour les indices i à j. Il nous servira de base pour comparer les autres lags. La variable *somme* (Ligne 14) est initialisée pour comparer les impacts des lags. La Ligne 16 teste s'il s'agit d'un déplacement d'instant d'exécution avant ou arrière. La Ligne 17 initialise la variable *section* dans le cas d'un déplacement d'instant d'exécution arrière, tandis que la Ligne 19 l'initialise pour le cas d'un déplacement d'instant d'exécution avant. En effet, il est possible de déterminer par rapport au lag à l'instant i si la valeur testée effectue un déplacement d'instant d'exécution avant ou arrière, ainsi seul le $lagsPlus1$ ou $lagsMoins1$ doit être testé.

La boucle Ligne 20 permet de calculer si l'impact du déplacement d'instant d'exécution approche l'ordonnancement d'un ordonnancement P-fair. Ainsi, la Ligne 21 permet de déterminer si la valeur absolue du lag de la variable $lagsPlus1$ (ou $lagsMoins1$) à l'indice k améliore le lag de base ou non. Cela permet ainsi d'évaluer, d'indice en indice, les améliorations d'un changement de lag. Les résultats sont ensuite retenus dans la variables *somme* (Ligne 22) où la somme à chaque indice est ajoutée.

Une fois que les résultats sont connus, la dernière étape est donc de les comparer afin de déterminer quel déplacement d'instant d'exécution obtient le meilleur impact pour se rapprocher le plus possible d'un ordonnancement P-fair avec un seul

déplacement d'instant d'exécution. Cela est fait grâce au test Ligne 23. Ainsi, si l'impact est amélioré, alors la variable $impactMax$ Ligne 24 enregistre cet intervalle dans les variables *début* Ligne 25 et fin Ligne 26.

Une fois toutes les boucles terminées et le meilleur impact sur le lag trouvé pour se rapprocher le plus de P-fair, les exécutions aux deux instants i et j sont inversées, Lignes 27 et 28 avant de retourner l'ordonnancement (Ligne 29).

Algorithme 12 : CalculLags

Entrées : booléen[] ordoVM, U^i, tmax
Sorties : double[] lags

```
1 double[] lags = double[tmax];
2 lags[0] = 0;
3 pour i = 1 ; i < tmax ; i = i + 1 faire
4     lags[i] = lags[i-1] + U^i - ordoVM[i-1];
5 retourner lags
```

Nous terminons la description des algorithmes par l'Algorithme 12. Il s'agit d'un algorithme appelé par l'Algorithme 11 à la Ligne 4. Cet algorithme permet ainsi de calculer les lags et de les sauvegarder dans un tableau. Il prend ainsi en entrée l'ordonnancement de la VM dont le lag doit être calculé $ordoVM$, ainsi que son utilisation U^i et la taille du tableau $tmax$. La sortie est le tableau des lags.

Les Lignes 1 et 2 permettent d'initialiser le tableau de lags à la taille $tmax$ et le premier lag du tableau $lags[0]$ à 0, comme défini par le lag.

La Ligne 3 correspond à la boucle qui permettra de calculer tous les lags suivants jusqu'à $tmax$. Ils sont ensuite calculés à la Ligne 4. Le calcul présenté ici correspond à celui donné lors du calcul itératif du lag (5.11). Le calcul est ainsi simplifié et évite de reparcourir pour chaque calcul l'ordonnancement entier de la VM étudiée. Une fois tous les lags calculés, ceux-ci sont ensuite retournés (Ligne 5).

Bien que cet algorithme itératif semble gourmand en calculs, il existe de nombreuses manières de l'optimiser en passant des étapes inutiles pour qu'il soit plus performant et obtienne un résultat en moins de temps.

5.3 Optimisations de l'algorithme itératif

Dans cette section, nous présentons différentes optimisations possibles pour améliorer le temps de calcul de l'algorithme itératif se rapprochant de P-fair. En effet, comme présenté dans la Section 5.2.2, l'algorithme peut paraître très gourmand en ressources, puisqu'il teste toute les combinaisons d'améliorations possibles afin de déterminer la meilleure.

Tout d'abord, il faut se rappeler que cet algorithme est utilisé de manière itérative, c'est-à-dire qu'il calcule l'ordonnancement des VMs une par une. Ainsi, plus le nombre de VMs déjà ordonnancées est élevé, plus cet algorithme sera rapide. Cette optimisation est déjà effectuée Lignes 8 et 11 dans l'Algorithme 11.

En effet, l'indice du début correspond à un instant où une des VMs déjà ordonnancées est exécutée, la boucle entière est passée, puisqu'une VM est déjà exécutée à cet instant, et que ces VMs ont déjà un ordonnancement trouvé, il n'est alors pas possible de changer ses instants d'exécution, nous ne pouvons changer que l'instant d'exécution de la VM étudiée. En effet, l'indice du début correspond à un instant

Temps	0	1	2	3	4	5	6	7	8	9	$\lvert Total \rvert$
Ordonnancement	0	0	0	1	0	1	1	1	0	0	
lags	0	0.4	0.8	1.2	0.6	1.0	0.4	-0.2	-0.8	-0.4	5.8
lagsMoins1	-1	-0.6	-0.2	0.2	-0.4	0.0	-0.6	-1.2	-1.8	-1.4	
somme			-0.6	-1.6	-1.8	-2.8	-2.6	-1.6	-0.6	0.4	
Ordonnancement	0	0	1	1	0	0	1	1	0	0	
lags	0	0.4	0.8	0.2	-0.4	0.0	0.4	-0.2	-0.8	-0.4	3.6

TABLE 5.2 : *lags* et *lagsMoins*1 d'une VM à ordonnancement donné avec 0.4 d'utilisation

où une des VMs déjà ordonnancées est exécutée, la boucle entière est passée. Puisqu'une VM est déjà exécutée à cet instant, et qu'il n'est pas possible de modifier cet instant d'exécution, nous pouvons alors passer la boucle entière. Il n'est possible de modifier que les instants d'exécution de la VM étudiée, et non celles déjà ordonnancées. Il est donc possible de passer directement à l'itération suivante. Ainsi, plus le nombre de VMs déjà ordonnancées augmente, plus l'algorithme pourra passer d'itérations. Ce comportement est le même lors de la seconde boucle, ce qui permet de trouver l'indice de fin : à chaque fois que l'indice correspond à un instant où une des VMs déjà ordonnancées est exécutée, nous passons à l'indice suivant.

Cela peut, par exemple, permettre lors de la dernière VM à ordonnancer, de n'avoir que très peu d'instants à tester et ainsi d'obtenir un algorithme extrêmement rapide, toutes les VMs étant déjà ordonnancées.

Il est également possible de trier les lags à tester. En effet, toutes les valeurs des lags ne sont pas nécessairement à tester. Cela est dû à l'incrémentation de 1 ou à la décrémentation de 1 du lag, elles-mêmes dues au déplacement des instants d'exécution des VMs.

Le but étant de réduire au plus la valeur du lag, si celui-ci est inférieur à -0.5, seul un déplacement d'instant d'exécution arrière pourra avoir des conséquences positives, celui-ci ajoutant 1 au lag et le rapprochant ainsi de 0. Réciproquement, seul un lag supérieur à 0.5 permet un déplacement d'instant d'exécution avant.

Ainsi chaque déplacement d'instant d'exécution ne peut être effectué que sous certaines conditions :

- Pour le déplacement arrière, le lag de *début* doit être inférieur à -0.5 et la VM ne doit pas être exécutée au temps *début*.
- Pour le déplacement avant, le lag de *début* doit être supérieur à 0.5 et la VM doit être exécutée au temps *début*.

Sans ces deux conditions pour chaque déplacement d'instant d'exécution, le déplacement d'instant d'exécution entier peut être omis, et tous les calculs des variables *section* et *section*1 peuvent être omis, aucune amélioration n'étant possible pour s'approcher d'un ordonnancement P-fair. Ainsi, à chaque tour de boucle Ligne 7, une seule section est testée. De plus, si $-0.5 < lag < 0.5$, la boucle entière Ligne 7 peut être omise, aucun déplacement d'instant d'exécution ne pouvant améliorer le lag.

Une autre possibilité d'optimisation peut être effectuée sur la boucle Ligne 10. En effet, celle-ci peut être arrêtée plus tôt et les calculs peuvent ne pas être utiles. Nous pouvons vérifier cela au cours de l'algorithme grâce à la variable *somme*. En effet, si elle devient positive, la section concernée devient alors invalide et il ne sert à rien de continuer la boucle Ligne 10. La somme permettant de déterminer si le déplacement d'instant d'exécution approche un ordonnancement P-fair en comparant

Temps	0	1	2	3	4	5	6	7	8	9	$\|Total\|$
Ordonnancement	1	1	1	0	0	1	1	1	0	1	
lags	0	-0.4	-0.8	-1.2	-0.6	0.0	-0.4	-0.8	-1.2	-0.6	6
lagsPlus1	1	0.6	0.2	-0.2	0.4	1.0	0.6	0.2	-0.2	0.4	
somme			-0.6	-1.6	-1.8	-0.8	-0.6	-1.2	-2.2	-2.4	
Ordonnancement	1	1	0	0	1	1	1	1	0	1	
lags	0	-0.4	-0.8	-0.2	0.4	0.0	-0.4	-0.8	-1.2	-0.6	4.8
Ordonnancement	1	1	0	0	0	1	1	1	1	1	
lags	0	-0.4	-0.8	-0.2	0.4	1.0	0.6	0.2	-0.2	-0.6	4.4

TABLE 5.3 : *lags* et *lagsPlus*1 d'une VM à ordonnancement donné avec une utilisation de 0.6

le lag de base à celui de la section, dès que la somme est positive, continuer le déplacement d'instant d'exécution éloignerait l'ordonnancement d'un ordonnancement P-fair, puisque le lag à chaque instant serait plus éloigné de 0. Continuer d'analyser cette section n'a alors plus d'intérêt et la section suivante peut alors être étudiée.

Nous proposons d'étudier les Tables 5.2 et 5.3 pour mettre en évidence l'importance de la variable *somme*. Dans ces tables, nous montrons sur la première ligne les instants d'exécution de la VM. Sur la ligne suivante, nous montrons le calcul du lag par rapport au premier ordonnancement donné. Sur la ligne suivante, nous montrons dans la Table 5.2 le résultat du tableau *lagsMoins*1 et dans la Table 5.3 le résultat du tableau *lagsPlus*1. Nous montrons ensuite les résultats des variables *somme* dans la Table 5.2 et 5.3 débutant au premier indice où il est important de calculer la somme, à savoir le premier indice où le lag est supérieur à 0.5 dans la Table 5.2 et le premier indice où le lag est inférieur à -0.5 pour la Table 5.3. Ce résultat est dû à l'optimisation précédente. Les lignes suivantes montrent le nouvel ordonnancement obtenu grâce à des déplacements d'instants d'exécution à partir de l'indice 2 pour les deux tables ainsi que le lag associé.

La dernière colonne de ces deux tables correspond à la valeur absolue de la somme totale des lags. Elle permet de mettre en avant si nous nous approchons d'un ordonnancement P-fair ou non : plus le résultat est proche de 0 et plus l'ordonnancement est proche d'un ordonnancement P-fair. Cette colonne est ainsi utilisée à chaque ligne où les lags sont présents.

Commençons par la Table 5.2. Dans cette table, nous montrons un ordonnancement déjà établi pour une VM avec une utilisation de 0.4 et son lag associé. La somme totale absolue du lag est ainsi de 5.8. D'après la variable *somme*3, la meilleure amélioration du lag est ainsi d'effectuer un déplacement d'instant d'exécution entre l'indice 2 et 5. Cela est d'autant plus justifié que le lag à l'instant 6 est inférieur à 0.5, celui-ci étant de 0.4. Nous montrons ensuite l'ordonnancement effectué après déplacement d'instant d'exécution et son lag associé. Cette fois, la somme totale absolue est de 3.6, ce qui nous rapproche donc d'un ordonnancement P-fair.

Dans la Table 5.3, nous cherchons à montrer l'importance d'effectuer le déplacement d'instant d'exécution sur les indices qui donnent à la variable *somme* le plus grand résultat et non d'uniquement se fier aux lags de référence qui peuvent revenir sur des valeurs qui semblent ne pas être intéressantes à échanger (comme par exemple 0 à l'instant 5 ou -0.4 à l'instant 6). Pour cela, nous commencerons par montrer un déplacement d'instant d'exécution qui est effectué sur des indices où les lags correspondent tous à des lags inférieurs à -0.5, à savoir des indices 2 à 4. Cela a pour effet d'obtenir une somme totale absolue de 4.8, contre 6 sur l'ordonnancement de départ. En effet, si la variable *somme* reste négative, le déplacement

d'instant d'exécution ne pourra pas approcher l'ordonnancement du modèle fluide P-fair.

Nous montrons également sur la Table 5.3 l'importance d'effectuer un déplacement d'instant d'exécution aux instants où la variable *somme* a la plus petite valeur. En effet, nous montrons ici que la plus petite valeur où le déplacement d'instant d'exécution est effectué est -2.2, à l'indice 8. C'est ce que nous montrons dans les deux dernières lignes : le déplacement d'instant d'exécution pris en compte est donc entre ces deux indices, 2 et 8, ce qui donne une somme totale absolue de 4.4 pour le lag. Ainsi, plus la somme totale absolue est basse, plus l'ordonnancement se rapprochera d'un ordonnancement P-fair.

Une autre possibilité d'optimisation est de passer la boucle Ligne 7. Cette fois, la condition pour passer la boucle dépend de l'indice précédent étudié. En effet, si par exemple l'indice 3 indique un lag, comme montré dans la Table 5.3 de -1.2 et que l'indice 2 indique un lag de -0.8, il est alors inutile de calculer la somme à partir de l'indice 3. En effet, un déplacement d'instant d'exécution à partir de l'indice 2 apportera de meilleurs résultats, puisque la variable *somme* à partir de cet indice aura un résultat plus petit et que nous cherchons à trouver le plus petit résultat. Nous distinguons ainsi deux types de lags : les lags inférieurs à -0.5 et les lags supérieurs à 0.5. Si deux lags pour des indices successifs sont du même type, il n'est alors pas nécessaire d'effectuer la boucle Ligne 7.

Nous proposons maintenant un récapitulatif de toutes les optimisations qu'il est possible d'effectuer pour arrêter des boucles entières ou pour passer à l'élément suivant d'une boucle. Récapitulons les optimisations sur la boucle Ligne 7 :

- Si une autre VM est ordonnancée que celle étudiée.
- Si le lag est supérieur à -0.5 et inférieur à 0.5.
- Si le lag à l'instant t est du même type que le lag à l'instant $t-1$, à savoir si $lag[t] < -0.5$ et $lag[t-1] < 0.5$ ou $lag[t] > 0.5$ et $lag[t-1] > 0.5$.

Récapitulons les optimisations sur la boucle Ligne 10 :

- Si une autre VM est ordonnancée que celle étudiée.
- Si la variable Ligne 14 devient positive.

Nous pouvons également envisager d'autres optimisations :

- Si le lag est inférieur à -0.5, seul un déplacement arrière est possible, la VM ne doit pas être exécutée à l'indice i et doit être exécutée à l'indice j, toutes les autres possibilités peuvent être omises.
- Si le lag est supérieur à 0.5, seul un déplacement avant est possible, la VM doit donc être exécutée à l'indice i et ne doit pas être exécutée à l'indice j, toutes les autres possibilités peuvent être omises.

Une dernière optimisation est également possible, mais celle-ci ne concerne pas directement l'algorithme. En effet, ici tous les calculs de lags proposés donnent des résultats décimaux, puisque le lag dépend de l'utilisation de la VM, entre 0 et 1. Ce type de calculs sur machine prennent beaucoup plus de temps que des calculs sur des entiers. Il serait ainsi possible de modifier le calcul du lag pour que celui-ci ne renvoie que des valeurs entières. Cela peut se faire facilement : l'utilisation de la VM est connue. Et celle-ci a la plupart du temps un nombre limité de chiffres après

la virgule. Il est alors possible d'identifier ce nombre n de chiffres, et d'obtenir un facteur multiplicateur $f = 10^n$. L'utilisation de la VM pourra ensuite également être multipliée par ce facteur f. Le calcul du lag dans l'Algorithme 12 à la Ligne 4 devient alors :

$$lags[i] = lags[i-1] + U^i \times f - ordoVM[i-1]$$

Où U^i a déjà été multiplié par le facteur f. Les modifications supplémentaires à apporter à l'Algorithme 11 sont moindres pour passer tous les calculs de lags en opérations entières : les variables $lagsPlus1$ et $lagsMoins1$ ne se font plus en ajoutant ou retirant 1 aux valeurs des lags, mais en ajoutant ou retirant f. Le seul autre changement concerne les optimisations déjà trouvées : les bornes ne seront plus -0.5 et 0.5 mais $-\frac{f}{2}$ et $\frac{f}{2}$.

À l'aide de toutes ces optimisations, l'algorithme itératif présenté peut omettre de nombreuses étapes et ainsi être bien plus rapide que prévu pour trouver un résultat. Toutes les optimisations ne sont pas présentées directement dans l'algorithme pour des raisons de lisibilité. Après avoir vu les différentes étapes générales de l'algorithme et comment l'optimiser, nous pouvons étudier les différentes manières de le combiner avec les algorithmes précédents pour ordonnancer les VMs et ainsi obtenir un plus grand nombre de VMs ordonnançables.

5.4 Combinaison des approches P-fair et harmonique

Dans cette section, nous proposons de voir les différentes possibilités pour utiliser les algorithmes présentés, que ce soit l'algorithme du Chapitre 4 basé sur une activation strictement périodique des VMs et des périodes d'activation harmoniques, ou les algorithmes présentés dans les Section 5.2.1 et Section 5.2.2. Ces algorithmes ont pour but de trouver un ordonnancement le plus proche d'un ordonnancement P-fair, soit de manière la plus optimale, soit en trouvant un compromis entre temps de calcul et résultat proche de P-fair, le tout de manière itérative, c'est-à-dire VM par VM.

En effet, grâce à ces différents algorithmes proposés et ces différentes approches, nous avons ainsi plusieurs façons de les combiner :

- Il est possible d'utiliser l'algorithme harmonique seul.
- Il est possible d'utiliser le problème d'optimisation linéaire seul.
- Il est possible d'utiliser l'algorithme proportionnel itératif seul.
- Il est possible de compléter l'algorithme harmonique avec le problème d'optimisation linéaire.
- Il est possible de compléter l'algorithme harmonique avec l'algorithme proportionnel itératif.

Cela permet ainsi d'avoir différentes approches, selon les besoins et surtout selon les jeux de tâches à ordonnancer au sein des VMs. Chaque approche peut ainsi offrir avantages et inconvénients. L'approche harmonique seule permet d'obtenir un résultat très rapidement, mais ne fonctionne en général que sur des jeux de tâches à paramètres particuliers, du fait de l'harmonie sur les périodes des VMs. En effet, l'harmonie ne peut fonctionner que lorsqu'il est possible de déterminer des paramètres harmoniques deux à deux pour les VMs, ce qui impose des jeux de tâches avec des périodes et des échéances particulières.

Pour le problème d'optimisation linéaire, celui-ci permettant d'obtenir un ordonnancement le plus proche de P-fair possible et prenant en compte les overheads, cette approche devrait être la plus utilisée. Mais celle-ci prenant beaucoup trop de temps pour trouver un résultat, il n'est pas possible de l'utiliser sur des systèmes réels en dehors de calculateurs dédiés.

L'algorithme proportionnel itératif propose en revanche un bon compromis entre le résultat s'approchant de P-fair et le temps de calcul. Il existe malgré tout plusieurs inconvénients à cette solution. Le premier étant que les overheads ne sont pas directement inclus. Il serait possible de les intégrer, mais cela ne devrait rester qu'une option à l'algorithme. En effet, le premier ordonnancement aléatoire trouvé par l'algorithme doit alors prendre en compte les overheads. Pour pouvoir les prendre en compte, il faut alors que le temps d'exécution à chaque exécution de la VM soit supérieur à une valeur définie par l'utilisateur. Par exemple, si l'utilisateur souhaite que l'overhead ne représente qu'au maximum 25% du temps d'exécution de la VM, chaque exécution doit être au minimum de 4 unités de temps. C'est ce temps minimal d'exécution que devra respecter le premier ordonnancement aléatoire trouvé. Les déplacements d'exécution ne pourront également se faire qu'en prenant en compte ce paramètre supplémentaire. Un tel comportement ne devrait être qu'une option, puisque cet algorithme est itératif, à savoir qu'il traite les VMs une par une. Ainsi, pour les dernières VMs, celles-ci devront être exécutées dans les espaces restants. Or, si ceux-ci sont insuffisants, l'algorithme ne parviendra pas à satisfaire les conditions sur les overheads.

Un autre inconvénient de cet algorithme est que, s'il est utilisé seul, les hyperpériodes générées seront grandes. Or, puisque cet algorithme créé un nombre d'exécution de la VM, qu'il n'est pas possible de déterminer à l'avance, le calcul du pire temps de réponse des tâches peut être complexe du fait du grand nombre de candidats d'instant critique.

Une autre raison qui pousserait à n'utiliser cette solution qu'en option est que l'algorithme proportionnel itératif a été proposé pour compléter l'algorithme harmonique, qui trouve en général un ordonnancement pour les premières VMs, mais pas les dernières. Le but était ainsi de créer un algorithme pour compléter cette solution, en s'approchant le plus possible de P-fair. L'utiliser en complétion de l'algorithme harmonique impose que des VMs soient déjà ordonnancées. L'algorithme proportionnel itératif devra alors trouver un ordonnancement pour les VMs suivantes dans les temps libres laissés par les VMs déjà ordonnancées. Puisque le but est de s'approcher d'un ordonnancement P-fair et que des VMs sont déjà ordonnancées, s'approcher d'un ordonnancement P-fair implique de compenser les temps où la VM ne peut pas être exécutée en l'exécutant de nombreuses fois d'affilée selon son utilisation. C'est pour cela que l'intégration des overheads sur cet algorithme ne devrait être qu'une option.

Nous montrons également, à travers des exemples, que ces approches ne sont pas comparables. Nous donnons ainsi quatre exemples à l'aide de deux jeux de tâches en testant les différentes possibilités, à savoir l'approche harmonique combinée à l'approche proportionnelle itérative et l'approche proportionnelle itérative seule. Ces différents exemples sont décrits à partir des Tables 5.4 et 5.5 qui présentent deux jeux de tâches différents, répartis en quatre VMs. Les paramètres des tâches sont ainsi définis dans chaque table respectivement, tout comme la VM dans laquelle chaque tâche doit être exécutée. Nous considérons, à travers ces exemples pour des raisons de lisibilité, que les overheads des VMs sont égaux à 0. Les algorithmes ne seront pas détaillés, mais les résultats seront donnés.

VM	Tâche	C_i	T_i	D_i
ρ^1	τ_1	1	20	11
ρ^2	τ_2	1	15	20
ρ^2	τ_3	2	15	30
ρ^3	τ_4	3	30	30
ρ^3	τ_5	4	40	80
ρ^4	τ_6	3	20	20

TABLE 5.4 : Jeu de tâches complet pour chaque VM du premier exemple

Exemple de l'approche harmonique combinée à l'approche proportionnelle itérative sur le jeu de tâches Table 5.4 : pour ce jeu de tâches, la période minimale est ici fixée à 4, puisque 4 VMs sont présentes dans le système et que l'overhead est fixé à 0. La période maximale est ici fixée à :

$$T^1_{max} \leq \left\lfloor \frac{11-1}{1-\frac{1}{20}} \right\rfloor$$
$$T^1_{max} \leq 10$$

Les deux paramètres $T^1_{min} = 4$ et $T^1_{max} = 10$ sont ainsi utilisés par l'Algorithme 2. Les résultats obtenus par cet algorithme sont $\rho^1(C^1 = 1, T^1 = 4)$. Les paramètres de la VM ρ^1 étant maintenant connus, il est possible d'utiliser l'Algorithme 3 pour trouver le plus de VMs qu'il est possible d'ordonnancer de manière harmonique. Cet algorithme trouve comme résultats pour la VM ρ^2 les paramètres $C^2 = 2$ et $T^2 = 8$. À titre informatif, la variable C^3_{test} trouvée est de 4, or puisque $T^1 - C^1 = 3$, ce paramètre n'est pas valable.

Nous utilisons ensuite l'Algorithme 4 pour trouver l'ordonnancement des VMs avec paramètres, à savoir ρ^1 et ρ^2. Ici l'ordonnancement est donné sur une hyper-période de 8, les VMs étant harmoniques. L'ordonnancement correspondant est : la VM ρ^1 est exécutée aux instants 0 à 1 et 4 à 5 et la VM ρ^2 aux instants 1 à 3.

Il ne reste maintenant plus qu'à utiliser l'Algorithme 9 pour trouver l'ordonnancement des deux autres VMs ρ^3 et ρ^4. Pour cela, l'algorithme a besoin en entrée des utilisations des deux VMs, ainsi que de l'ordonnancement déjà obtenu pour les autres VMs et du temps maximum sur lequel l'ordonnancement doit être trouvé, correspondant à l'hyper-période des VMs et des tâches.

Commençons par calculer les utilisations des VMs. Tout d'abord, nous devons connaître l'utilisation des VMs déjà ordonnancées, à savoir $\frac{1}{4} = 0.25$ pour ρ^1 et $\frac{2}{8} = 0.25$ pour ρ^2. L'utilisation du jeu de tâches de la VM ρ^3_τ est égale à $\frac{3}{30} + \frac{4}{40} = \frac{1}{5}$. L'utilisation du jeu de tâches de la VM ρ^4_τ est égale à $\frac{3}{20}$.

Ainsi, d'après l'Équation 5.14, U^3 vaut :

$$\begin{aligned} U^3 &= \frac{U^3_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1 - \sum_{\rho^k \in \rho_o} U^k) \\ &= \frac{\frac{1}{5}}{\frac{1}{5} + \frac{3}{20}} \times (1 - \frac{1}{4} - \frac{2}{8}) \\ &= \frac{2}{7} \end{aligned}$$

Et U^4 vaut :

$$\begin{aligned} U^4 &= \frac{U^4_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1 - \sum_{\rho^k \in \rho_o} U^k) \\ &= \frac{\frac{3}{20}}{\frac{1}{5} + \frac{3}{20}} \times (1 - \frac{1}{4} - \frac{2}{8}) \\ &= \frac{3}{14} \end{aligned}$$

Un dernier paramètre reste à déterminer en entrée de l'algorithme, le temps maximal sur lequel l'ordonnancement doit être trouvé. Les utilisations étant connues, nous pouvons alors utiliser le plus petit commun multiple entre les périodes des VMs avec paramètres et les périodes virtuelles (à savoir le dénominateur dans la forme fractionnaire de l'utilisation).

Ainsi le dernier paramètre inconnu $tmax$ est :

$$tmax = ppcm(4, 8, 7, 14) = 56$$

Où $ppcm$ est la fonction permettant de calculer le plus petit commun multiple.

Avec ces utilisations données aux VMs, en s'approchant d'un ordonnancement P-fair à l'aide de l'Algorithme 9, l'ordonnancement donné aux VMs fait que toutes les tâches de ρ^3 et ρ^4 respectent leurs échéances. L'exemple n'est pas donné en entier pour des raisons de lisibilité, mais les échéances sont bien respectées.

Exemple de l'approche proportionnelle itérative seule sur le jeu de tâches Table 5.4 : Reprenons les jeux de tâches de la Table 5.4, mais en n'utilisant cette fois que l'approche proportionnelle itérative et non l'algorithme harmonique. Commençons donc par les calculs d'utilisations :

$$\begin{aligned} U^1 &= \frac{U^1_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1 - 0) \\ &= \frac{\frac{1}{20}}{\frac{1}{20} + \frac{1}{5} + \frac{1}{5} + \frac{3}{20}} \\ &= \frac{1}{12} \end{aligned}$$

$$\begin{aligned} U^2 &= \frac{U^1_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1 - 0) \\ &= \frac{\frac{1}{5}}{\frac{1}{20} + \frac{1}{5} + \frac{1}{5} + \frac{3}{20}} \\ &= \frac{1}{3} \end{aligned}$$

VM	Tâche	C_i	T_i	D_i
ρ^1	τ_1	8	20	15
ρ^2	τ_2	1	15	20
ρ^2	τ_3	2	15	30
ρ^3	τ_4	3	50	50
ρ^3	τ_5	1	25	35
ρ^4	τ_6	1	20	20

TABLE 5.5 : Jeu de tâches complet pour chaque VM du second exemple

$$\begin{aligned} U^3 &= \frac{U^1_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1-0) \\ &= \frac{\frac{1}{5}}{\frac{1}{20}+\frac{1}{5}+\frac{1}{5}+\frac{3}{20}} \\ &= \frac{1}{3} \end{aligned}$$

$$\begin{aligned} U^4 &= \frac{U^1_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1-0) \\ &= \frac{\frac{3}{20}}{\frac{1}{20}+\frac{1}{5}+\frac{1}{5}+\frac{3}{20}} \\ &= \frac{1}{4} \end{aligned}$$

Nous obtenons ainsi ces quatre utilisations : $U^1 = \frac{1}{12}$, $U^2 = \frac{1}{3}$, $U^3 = \frac{1}{3}$ et $U^4 = \frac{1}{4}$. Le $tmax$ obtenu est ainsi de :

$$tmax = ppcm(12, 3, 4) = 12$$

Commençons par l'ordonnancement donné à ρ^1 par l'Algorithme 9. Peu importe l'ordonnancement trouvé, cette VM ne pourra être ordonnancée qu'une seule fois toutes les 12 unités de temps, l'hyper-période étant de 12 et son utilisation de $\frac{1}{12}$. La tâche $\tau_1(1, 20, 11)$ ne peut donc pas respecter son échéance, celle-ci étant de 11 et ne pouvant être exécutée que toutes les 12 unités de temps.

Exemple de l'approche proportionnelle itérative seule sur le jeu de tâches Table 5.5 : Pour les exemples suivants, nous utilisons le jeu de tâches donné par la Table 5.5. Pour y appliquer l'algorithme proportionnel itératif, il nous faut alors les utilisations des VMs. Pour commencer, ils nous faut l'utilisation du jeu de tâches associé à chaque VM. Nous obtenons ainsi : $U^1_\tau = \frac{8}{20}$, $U^2_\tau = \frac{1}{15} + \frac{2}{15} = \frac{1}{5}$, $U^3_\tau = \frac{3}{50} + \frac{1}{25} = \frac{1}{10}$ et $U^4 = \frac{1}{20}$. Comme précédemment, nous calculons les utilisations données aux VMs à l'aide de l'Équation 5.14 :

$$
\begin{aligned}
U^1 &= \frac{U^1_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1-0) \\
&= \frac{\frac{8}{20}}{\frac{8}{20} + \frac{1}{5} + \frac{1}{10} + \frac{1}{20}} \\
&= \frac{8}{15}
\end{aligned}
$$

$$
\begin{aligned}
U^2 &= \frac{U^1_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1-0) \\
&= \frac{\frac{1}{5}}{\frac{8}{20} + \frac{1}{5} + \frac{1}{10} + \frac{1}{20}} \\
&= \frac{4}{15}
\end{aligned}
$$

$$
\begin{aligned}
U^3 &= \frac{U^1_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1-0) \\
&= \frac{\frac{1}{10}}{\frac{8}{20} + \frac{1}{5} + \frac{1}{10} + \frac{1}{20}} \\
&= \frac{4}{30}
\end{aligned}
$$

$$
\begin{aligned}
U^4 &= \frac{U^1_\tau}{\sum\limits_{\rho^k \in \rho_n} U^k_\tau} \times (1-0) \\
&= \frac{\frac{1}{20}}{\frac{8}{20} + \frac{1}{5} + \frac{1}{10} + \frac{1}{20}} \\
&= \frac{1}{15}
\end{aligned}
$$

Dans ce cas, $tmax$ est égal à :

$$tmax = ppcm(15, 30) = 30$$

Nous pouvons maintenant tester l'Algorithme 9. Pour des raisons de place, et l'algorithme commençant par trouver un ordonnancement aléatoire, nous ne le déroulerons pas ici, mais nous présentons le résultat de cet algorithme dans la Figure 5.1.

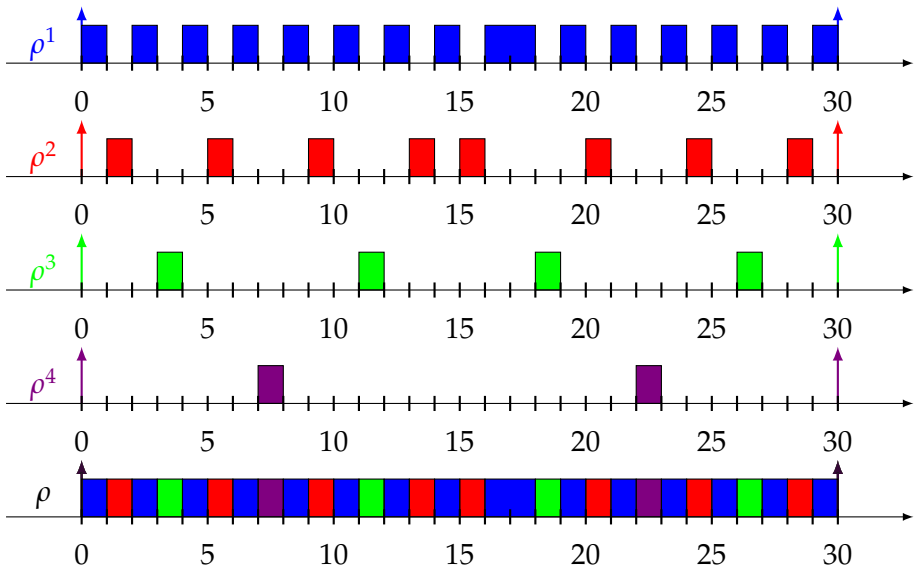


FIGURE 5.1 : Ordonnancement approchant de P-fair donné par l'Algorithme 9 d'après le jeu de tâches de la Table 5.5

Comme nous le voyons dans cette figure, s'approcher d'un ordonnancement P-fair de manière itérative permet également de s'approcher d'un ordonnancement P-fair pour toutes les VMs. Le nombre de candidats d'instant critique étant élevé pour chaque VM, nous ne montrerons pas les calculs de temps de réponse pour chaque tâche dans chaque VM, mais un tel ordonnancement garantit bien le respect de toutes les échéances des tâches.

Exemple de l'approche harmonique seule sur le jeu de tâches Table 5.5 : Dans ce dernier exemple, nous cherchons à utiliser l'approche harmonique sur le même jeu de tâches de la Table 5.5 sur lequel l'approche proportionnelle itérative a pu trouver un résultat. Nous devons donc utiliser l'Algorithme 2.

Pour cela, nous avons besoins de T^1_{min} et T^1_{max}. L'overhead étant à 0 et le système contenant quatre VMs, T^1_{min} est donc de 4. La période maximale est calculée ainsi :

$$T^1_{max} \leq \left\lfloor \frac{15-8}{1-\frac{8}{20}} \right\rfloor$$
$$T^1_{max} \leq 11$$

Nous pouvons ensuite utiliser l'Algorithme 2. Le problème ici est qu'avec de telles périodes minimales et maximales, l'algorithme ne peut pas trouver de paramètres pour la VM ρ^1 qui respecte l'échéance de la tâche τ_1. En effet, l'algorithme teste d'abord une première fois avec les paramètres $\rho^1(2,4)$, ce qui donne un temps de réponse de 16 unités de temps pour τ_1. L'itération suivante teste avec les paramètres $\rho^1(3,7)$, ce qui donne un temps de réponse de 20. L'itération suivante teste $\rho^1(4,9)$ et le temps de réponse est de 18, et enfin la dernière itération teste les paramètres $\rho^1(4,10)$, ce qui donne un temps de réponse de 20. L'algorithme 2 ne peut ainsi pas trouver de paramètres convenables pour la VM ρ^1.

À travers tous ces exemples nous avons réussi à montrer que ces algorithmes ne sont pas comparables. En effet, pour le jeu de tâches de la Table 5.4, l'algorithme harmonique complété par l'approche proportionnelle itérative peut ordonnancer le jeu de tâches alors que l'approche proportionnelle itérative seule ne peut l'ordonnancer.

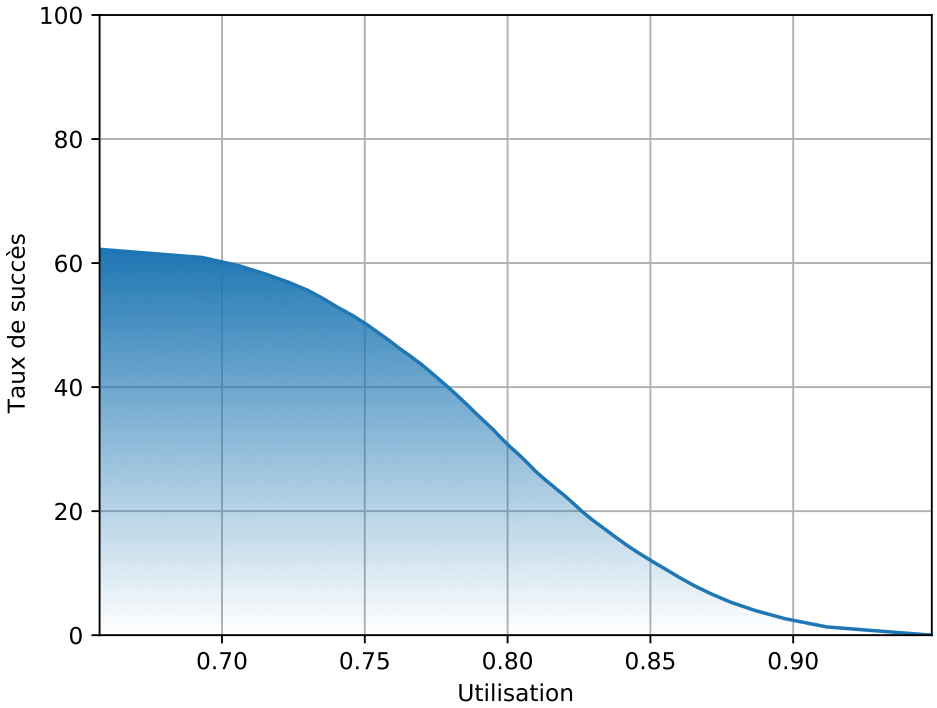


FIGURE 5.2 : Taux de succès de la combinaison des algorithmes proportionnels itératifs et harmoniques en fonction de l'utilisation

Nous avons également montré à l'aide de la Table 5.5 que sur ce système, seul l'approche proportionnelle itérative permet de respecter les échéances des tâches alors que l'approche harmonique ne le permet pas.

Ainsi, aucun algorithme ne surpasse l'autre, et nous avons ainsi proposé différentes approches pour ordonnancer le plus de systèmes possibles. Pour un système donné, il est ainsi possible de tester les différentes approches pour déterminer si l'une d'elles peut répondre aux contraintes temporelles des tâches.

Une dernière étape reste nécessaire pour vérifier l'efficacité de la combinaison des algorithmes : des simulations pour évaluer le taux de succès de la combinaison de l'algorithme proportionnel itératif sur les VMs non ordonnancées par l'algorithme harmonique. Pour cela, nous reprenons les mêmes simulations et la même génération de tâches, ainsi que les mêmes systèmes, tels que définis dans les simulations de la Section 4.4.

Il s'agit donc de systèmes à quatre VMs où sont répartis quatre jeux de tâches. Leur utilisation est fixée entre 0.1 et 0.2 pour les trois premiers et entre 0.3 et 0.4 pour le dernier. Plus de détails sur la génération des tâches sont donnés dans la Section 7.1 et plus de détails sur l'algorithme utilisé pour les simulations sont donnés dans la Section 7.2 du Chapitre 7. Les résultats sont ainsi donnés dans la Figure 5.2.

Dans cette figure, nous obtenons un taux de succès décroissant, débutant à un peu plus de 60% de jeux de tâches ordonnançables et décroissant pour s'approcher de 0% en approchant une utilisation de 1. Pour des systèmes à quatre VMs et avec une telle charge, les résultats peuvent être considérés comme satisfaisants, mais nous avons cherché à augmenter encore ce taux de succès.

Pour cela, nous proposons de nouveaux algorithmes au second niveau hiérarchique pour permettre d'ordonnancer encore plus de systèmes avec ces simples approches. Il sera ainsi possible de combiner ces différents algorithmes d'ordonnancement au premier niveau, avec en plus un nouvel algorithme d'ordonnancement pour le second niveau : l'algorithme d'ordonnancement Dual Priority.

Chapitre 6

Amélioration de l'ordonnançabilité des tâches dans des VMs hypervisées : Dual Priority (en agissant sur le second niveau uniquement)

Dans ce chapitre, nous montrons une nouvelle façon d'ordonnancer les tâches au second niveau. Nous utilisons ici Dual Priority. Elle permet d'aller plus loin que la priorité fixe, qui n'a qu'une seule priorité et a des propriétés proches de celles d'Earliest Deadline First (EDF). En effet, au niveau des tâches, deux priorités sont alors utilisées pour chaque tâche, et une échéance intermédiaire est définie pour savoir quand passer d'une bande de priorité à l'autre. Habituellement, Dual Priority est utilisée dans un contexte sans hypervision. Nous étudions d'abord ce contexte avant de l'amener vers un modèle hypervisé hiérarchique à deux niveaux.

Pour cela, nous utilisons dans ce chapitre un contexte de tâches synchrones périodiques. En effet, pour Dual Priority, le pire cas n'est pas connu. C'est donc ce contexte qui sera retenu, celui-ci étant un candidat d'instant critique dans notre modèle pour RM (voir Définition 42 du Chapitre 3). Pour rappel, les candidats sont quand toutes les tâches sont synchrones puis lancées de manière périodique à la fin d'une exécution de la VM.

Concernant la partie sur Dual Priority sans modèle hiérarchique, nous commençons par définir tous les concepts pour l'ordonnancement Dual Priority ainsi que son algorithme le plus utilisé pour définir les nouveaux paramètres des tâches (les deux priorités et l'échéance intermédiaire) : First Deadline Miss Strategy (FDMS). Cet algorithme étant gourmand en ressources de calcul et en mémoire pour obtenir un résultat, nous proposons également une autre approche, bien plus rapide pour calculer les paramètres des tâches mais avec un taux de succès moins élevé, à savoir Reverse Rate Monotonic (RM^{-1})/RM. Nous proposons ensuite d'y intégrer les overheads avant de voir les gains obtenus grâce à Dual Priority en l'intégrant dans un modèle hiérarchique à deux niveaux.

Sommaire

6.1 Concept de Dual Priority

Dans cette section, nous commençons par définir plus formellement l'algorithme Dual Priority. Nous définissons également les bases et quelques définitions associées à cet algorithme, qui impose un nouveau modèle de tâches. Nous expliquons également en quoi cet algorithme peut être important et pourquoi celui-ci sera utilisé au cours de la thèse.

Tout d'abord, commençons par présenter le modèle de tâches associé à l'algorithme Dual Priority. Ce modèle a été présenté pour la première fois dans [31] pour répondre à un problème : l'optimalité des algorithmes d'ordonnancement préemptifs à priorité fixe au niveau des tâches est restreinte par rapport aux algorithmes d'ordonnancement à priorité fixe au niveau des travaux (voir Chapitre 2). En particulier, dans le cas de tâches à échéances sur requête, l'ordonnancemement RM, bien qu'optimal n'atteint pas le niveau d'ordonnançabilité d'EDF (100%).

Une condition seulement suffisante existe pour RM (voir état de l'art Chapitre 2) :

$$U \leq n \times \left(2^{\frac{1}{n}} - 1\right)$$

Cette condition suffisante permet seulement de conclure que tout jeu de tâches ayant une utilisation inférieure à ln(2)>0.69 est ordonnançable avec RM. On ne peut pas conclure sur l'ordonnançabilité du jeu de tâches si $ln(2) < U \leq 1$.

Comme présenté dans l'état de l'art (Chapitre 2, Section 2.2), il existe un seuil moyen d'ordonnançabilité pour RM situé à une utilisation de 0.88.

Les ordonnanceurs à priorité fixe au niveau des tâches sont simples et moins coûteux : il suffit de déterminer, en fonction de la priorité assignée à la tâche, a priori quelle instance de tâche exécuter, celle-ci étant connue.

C'est à partir de ce constat que les recherches sur l'algorithme Dual Priority ont débuté. En effet, le but de cette classe d'algorithme est de gagner en ordonnançabilité par rapport aux algorithmes à priorité fixe au niveau des tâches en proposant un ordonnancement qui n'utilise non plus une seule priorité fixe au niveau des tâches mais deux. Pour cela, il est également nécessaire d'ajouter un autre paramètre : une échéance intermédiaire de promotion pour passer d'une priorité fixe à l'autre.

Pour l'ordonnancement Dual Priority, une tâche τ_i se définit ainsi :

$$\tau_i(C_i, T_i, D_i, P_i^1, P_i^2, S_i)$$

Où les premiers paramètres C_i, T_i et D_i sont les paramètres classiques d'une tâche temps réel (voir Chapitre 2), respectivement le WCET, la période et l'échéance relative de la tâche. En revanche, les nouveaux paramètres sont les deux nouvelles priorités fixes, à savoir P_i^1 et P_i^2. Un nouveau paramètre fait également son apparition : il s'agit de S_i. Ce paramètre représente une échéance intermédiaire relative à la date d'activation également appelée échéance de promotion à partir de laquelle l'instance de la tâche change de priorité et passe de la priorité P_i^1 à P_i^2. La priorité d'une tâche est d'autant plus grande que sa valeur est faible (la plus haute priorité correspond à la valeur 1).

Nous définissons ainsi deux priorités pour une tâche τ_i. Une première, de priorité basse, égale à P_i^1 et une de haute priorité égale à P_i^2 de telle sorte que :

$$\forall \tau_i \in \tau, P_i^1 < P_i^2$$

Tâche	C_i	T_i	D_i	P_i^1	P_i^2	S_i
τ_1	3	6	6	4	1	6
τ_2	2	8	8	5	2	8
τ_3	3	12	12	6	3	10

TABLE 6.1 : Jeu de tâches complet pour Dual Priority

Concernant l'échéance intermédiaire S_i d'une tâche τ_i, celle-ci est définie par :

$$0 \leq S_i \leq D_i$$

Nous supposons également que les priorités avant et après promotion sont affectées par bandes de priorité, i.e. toute priorité avant promotion est inférieure à toute priorité après promotion :

$$\max_{\tau_i \in \tau} P_i^1 < \min_{\tau_i \in \tau} P_i^2$$

Notons qu'une tâche peut être directement ordonnancée avec sa seconde priorité P_i^2 lorsque $S_i = 0$ ou celle-ci peut être ordonnancée uniquement à sa première priorité P_i^1 lorsque $S_i = D_i$. Cela revient à une tâche à une seule priorité dans les deux cas.

Concernant l'implémentation d'une telle classe d'algorithme, deux solutions sont proposées : soit le noyau du Real-Time Operating System (RTOS) propose directement d'avoir deux priorités pour les tâches, soit il doit pouvoir changer les priorités des tâches en cours d'exécution. Ces deux implémentations restent plus légères qu'une implémentation d'EDF. En effet, pour l'algorithme Dual Priority, un seul timer est nécessaire pour gérer le basculement d'une priorité fixe à une autre.

En revanche, l'algorithme EDF a de nombreux inconvénients qui ne sont pas visibles du point de vue théorique. En effet, pour commencer, un système avec ordonnanceur à priorité fixe au niveau des instances de tâches doit contenir plus d'informations et est donc plus gourmand en mémoire. À chaque activation d'une instance, la priorité d'une instance de tâche, égale à son échéance absolue, doit être définie. En fonction du nombre de bits alloués au codage de l'échéance absolue définie en référence à un instant 0, origine des temps, il peut être nécessaire de changer l'origine des temps pour que la valeur des échéances absolues reste codable. Cela revient, si l'opération est faite à un instant t, à retirer à l'ensemble des instances en cours t à la valeur des échéances absolues. Lorsque l'opération est trop souvent répétée, cela peut devenir coûteux en temps de calcul.

Nous commençons par montrer les avantages d'un tel type de l'ordonnancement Dual Priority à travers un exemple. Le jeu de tâches est défini dans la Table 6.1. À partir de ce système, nous présentons deux ordonnancements différents : un premier, utilisant RM (Figure 6.1) et un second, reprenant les paramètres de la Table 6.1 pour l'ordonnancement Dual Priority (Figure 6.2).

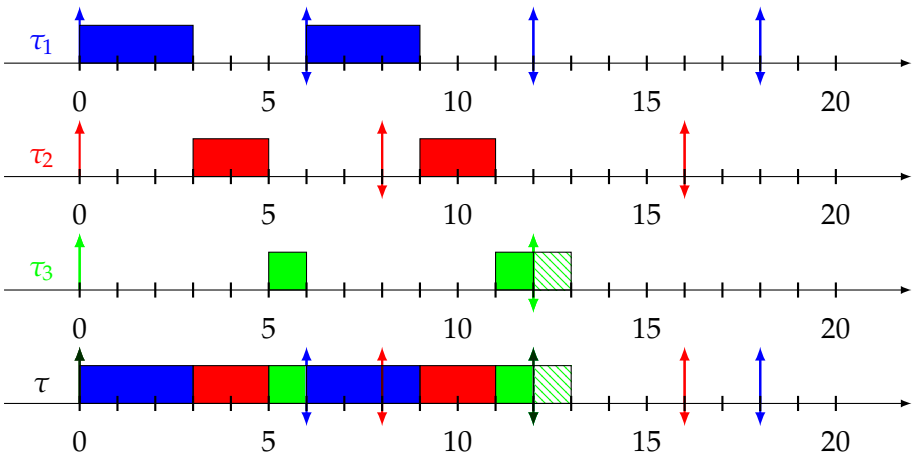


FIGURE 6.1 : Ordonnancement du jeu de tâches de la Table 6.1 en RM

Dans la première figure, nous montrons que l'ordonnancement avec RM échoue à l'instant 12 pour la tâche τ_3. En effet, celle-ci n'a pu être exécutée que deux fois alors que son WCET est de 3. Ainsi, l'ordonnancement RM ne peut pas respecter les échéances de toutes les tâches pour un tel système.

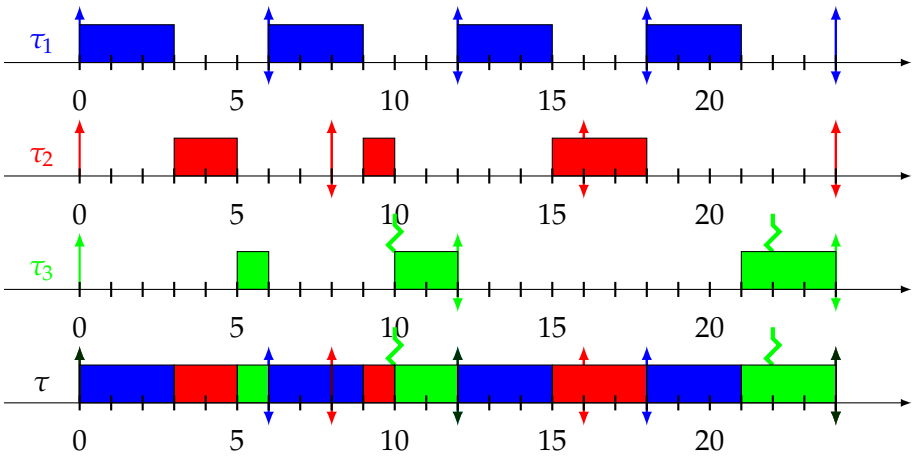


FIGURE 6.2 : Ordonnancement du jeu de tâches de la Table 6.1 en Dual Priority

En revanche, si nous considérons Dual Priority, comme dans la Figure 6.2, nous pouvons voir que le jeu de tâches respecte ses échéances. Le jeu de tâches décrit dans la Table 6.1 donne ainsi une échéance intermédiaire S_3 pour la tâche τ_3 de 10. Les autres tâches ont un S_i égal à la période T_i, ces deux tâches n'ont donc en réalité qu'une seule priorité, le passage à la priorité la plus haute étant au même instant où une nouvelle instance de tâche est créée. Nous montrons les échéances intermédiaires S_3 de la tâche τ_3 dans la Figure 6.2 aux instants 10 et 22 à l'aide d'une ligne brisée. Nous utilisons une ligne brisée pour représenter les échéances intermédiaires, si besoin, à travers les prochaines figures. Nous montrons dans la Figure 6.2 une exécution complète du jeu de tâches sur une durée égale à l'hyper-période du jeu de tâches (24). Nous montrons ainsi que grâce à ce simple changement (ajouter une échéance intermédiaire à τ_3 de 10), le jeu de tâches devient ordonnançable avec Dual Priority.

Nous montrons ainsi à travers ces deux exemples l'intérêt de Dual Priority par rapport à RM : il permet d'ordonnancer des jeux de tâches à utilisation plus élevée,

allant jusqu'à 100%. Dès la première présentation de Dual Priority en 1993, il était conjecturé que Dual Priority pouvait ordonnancer n'importe quel jeu de tâches à échéances sur requêtes ayant une utilisation inférieure ou égale à 100% ([31]). Cette conjecture a été prouvée pour un jeu de tâches de deux tâches en 2010 ([32]). Nous verrons par la suite que cette conjecture a été depuis peu réfutée pour un nombre de tâches plus grand que deux à l'aide d'un contre-exemple montré dans [54].

Pour comparer Dual Priority, nous introduisons une classe d'algorithme plus générale que Dual Priority appelée m-Priority ([67]). Il s'agit non plus de n'avoir que deux priorités fixes au niveau des tâches mais m priorités fixes au niveau des tâches.

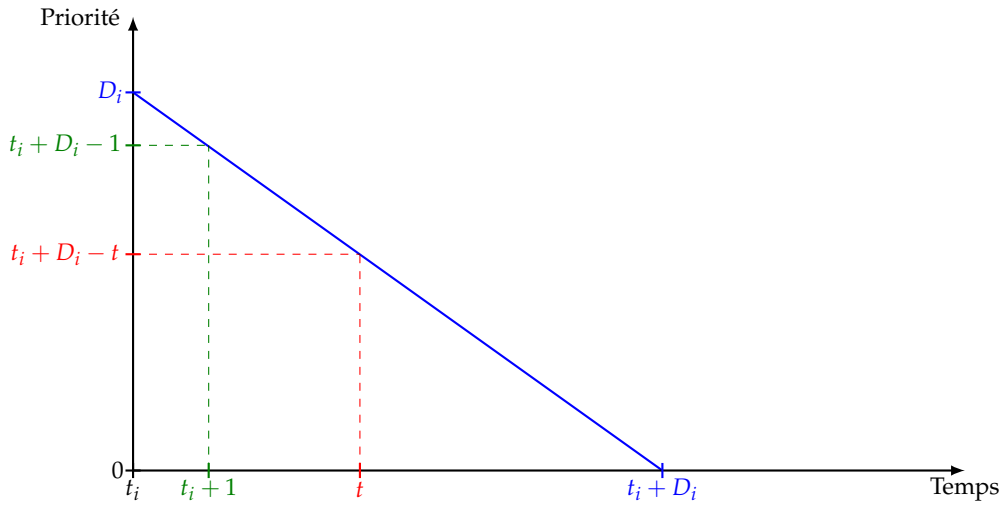


FIGURE 6.3 : Transformation des priorités d'une tâche τ_i activée à l'instant t_i pour correspondre aux m-priorités de EDF

Nous montrons dans la Figure 6.3 comment transformer EDF en un algorithme à $max_{\tau_i \in \tau}(D_i)$ priorités fixes au niveau des tâches. Dans cette figure, une instance de la tâche τ_i est activée à un instant t_i. La priorité donnée par EDF pour cette instance correspondant à son échéance absolue $t_i + D_i$. La priorité fixe attribuée à cette instance à un instant t est alors $t_i + D_i - t$. Si cette instance est maintenant ordonnancée par un algorithme m-Priority, avec D_i priorités fixes pour l'instance de τ_i de telle sorte que sa priorité fixe soit égale à D_i à son activation, et que sa priorité fixe soit décrémentée de 1 à chaque unité de temps, alors l'ordonnancement obtenu est équivalent à EDF. En effet, pour deux instances des tâches τ_i et τ_j à tout instant t, $t_i + D_i < t_j + D_j \Leftrightarrow t_i + D_i - t < t_j + D_j - t$. Ainsi l'ordonnancement obtenu par EDF est strictement équivalent à l'ordonnancement obtenu par un algorithme $max_{\tau_i \in \tau}(D_i) - Priority$.

En effet, si la plus grande échéance relative est connue, il est alors possible avec un ordonnancement $max_{\tau_i \in \tau}(D_i) - Priority$ de reproduire un ordonnancement EDF. Cependant, faire évoluer la priorité fixe des travaux selon l'algorithme $max_{\tau_i \in \tau}(D_i) - Priority$ à chaque tick d'horloge n'est pas nécessaire tant qu'une nouvelle instance de tâche n'est pas activée. Nous nous posons alors cette question : «est-il possible de voir EDF comme un cas particulier d'algorithme Dual Priority ?». Ce qui reviendrait à montrer que pour toute instance de tâches, au maximum un seul changement de priorité est nécessaire avec l'ordonnancement $max_{\tau_i \in \tau}(D_i) - Priority$ pour reproduire le comportement d'EDF.

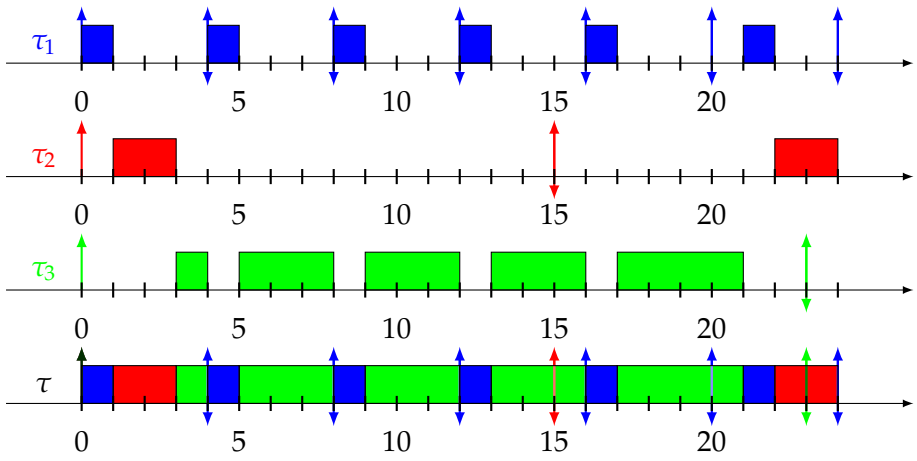


FIGURE 6.4 : Ordonnancement du jeu de tâches $\tau_1(1,4,4)$, $\tau_2(2,15,15)$ et $\tau_3(14,23)$ avec deux promotions

Nous montrons dans la Figure 6.4 qu'EDF ne peut pas être traduit en un algorithme Dual Priority mais doit être traduit en un algorithme 3-Priority, conduisant à deux promotions au minimum pour être ordonnançable.

En effet, comme nous pouvons le voir, la tâche τ_3 a une priorité inférieure à celle de τ_1 et τ_2 jusqu'à l'instant 15. De l'instant 15 à 20, la priorité de τ_3 est uniquement inférieure à celle de τ_1 et est supérieure à celle de τ_2. En revanche, de l'instant 20 à 23 la tâche τ_3 a la plus grande priorité. Ainsi, pour qu'un ordonnancement $m-$ *Priority* soit équivalent à EDF, il faudrait au moins trois priorités différentes et deux promotions. Ce qui permet de conclure qu'il n'est pas toujours possible de traduire EDF en un algorithme Dual Priority.

Concernant Dual Priority, deux problèmes restent ensuite à déterminer : comment choisir les différentes priorités et quelle valeur donner aux échéances intermédiaires. Une première méthode, non formelle, a d'abord été proposée pour répondre à ce problème. Pour résumer, les priorités sont assignées dans l'ordre RM pour la bande basse et dans l'ordre RM pour la bande haute.

Concernant l'échéance intermédiaire, celle-ci est trouvée de manière itérative : le jeu de tâches est d'abord initialisé avec des échéances intermédiaires égales à la période pour toutes les tâches du jeu de tâches. Une simulation de l'ordonnancement ainsi obtenu est ensuite faite jusqu'à ce qu'une échéance soit manquée. La tâche pour laquelle l'échéance est manquée voit ainsi son échéance intermédiaire diminuer jusqu'à ce que l'instance soit ordonnançable. Les simulations sont ensuite itérées avec ces nouveaux paramètres jusqu'à ce qu'aucune échéance ne soit manquée ou qu'il ne soit pas possible de diminuer davantage l'échéance arbitraire (donc $S_i = 0$). Cette méthode sera ensuite formalisée dans la Section 6.2 sous le nom de FDMS.

En 2019, [54] montre que l'algorithme Dual Priority n'est pas optimal dans le cas général. Différents exemples ont été trouvés de manière exhaustive en testant sur chaque exemple toutes les possibilités d'assignements de priorités ainsi que toutes les possibilités d'échéances arbitraires. Trois exemples sont relevés avec des propriétés différentes :

- Un premier jeu de tâches décrit par la Table 6.2 n'a aucune attribution de paramètres pour être ordonnançable par Dual Priority.
- Un second jeu de tâches pour lequel il n'existe aucun assignement de priorités RM pour la première bande de priorité qui rend le jeu de tâches ordonnançable (Table 6.3).

Tâche	C_i	T_i	D_i
τ_1	8	19	19
τ_2	13	29	29
τ_3	9	151	151
τ_4	14	197	197

TABLE 6.2 : Jeu de tâches dont aucun paramètre ne peut être ordonnançable en utilisant Dual Priority

Tâche	C_i	T_i	D_i		P_i^1	P_i^2	S_i
τ_1	13	29	29		4	0	13
τ_2	17	47	47		5	1	17
τ_3	4	89	89		7	2	42
τ_4	28	193	193		6	3	139

TABLE 6.3 : Jeu de tâches ordonnançable en Dual Priority mais dont aucun assignement de priorités effectué par RM pour la première bande de priorités n'est ordonnançable

- Un troisième jeu de tâches pour lequel il existe un assignement de priorités où RM est utilisé pour les deux bandes de priorités mais qui n'est pas ordonnançable en utilisant FDMS (Table 6.4).

[54] montre également que la procédure FDMS n'est pas optimale pour l'attribution des échéances intermédiaires de promotion. Pour trouver des contre exemples, il a fallu partir de jeux de tâches aléatoires sur lesquels toutes les possibilités d'assignement de priorités et toutes les échéances intermédiaires ont été testées. Étant donné la complexité et le temps de calculs pour trouver ces exemples, cela explique que la conjecture sur l'optimalité de Dual Priority soit restée sans réponse pendant des années. Un autre facteur joue également : les jeux de tâches permettant de réfuter l'optimalité de Dual Priority sont extrêmement rares. On parle alors de quasi-optimalité de Dual Priority.

D'autres propriétés contre-intuitives ont en revanche été vérifiées. Parmi ces propriétés, certaines rendent une analyse de ce modèle difficile en dehors des simulations. La première propriété concerne les candidats d'instant critique (comme défini dans la Section 3.2 du Chapitre 3).

Propriété 2 (Candidat d'instant critique synchrone)
Pour un jeu de tâches donné, le scénario synchrone ne donne pas nécessairement un candidat d'instant critique. Il n'est ainsi pas possible de déterminer des candidats d'instant critique.

Tâche	C_i	T_i	D_i		P_i^1	P_i^2	S_i
τ_1	6	11	1		4	0	5
τ_2	6	20	20		5	1	3
τ_3	4	46	46		6	2	24
τ_4	5	74	74		7	3	35

TABLE 6.4 : Jeu de tâches ordonnançable avec un assignement de priorités RM pour la première bande de priorité et la seconde mais dont les paramètres ne peuvent pas être trouvés par FDMS

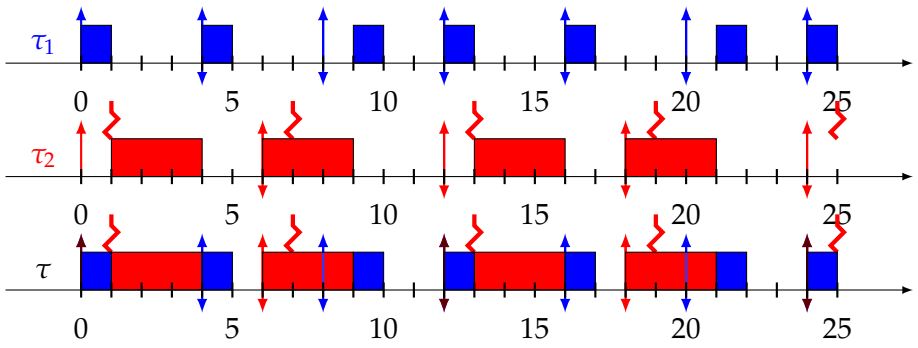
Propriété 3 (Temps de réponse de la première instance de tâche)
La première instance de tâche ne donne pas nécessairement le temps de réponse le plus élevé dans un scénario synchrone.

Définition 55 (Période d'étude)
La période d'étude d'un système correspond au temps pendant lequel le système doit être étudié pendant une simulation pour déterminer si toutes les échéances des tâches sont respectées.

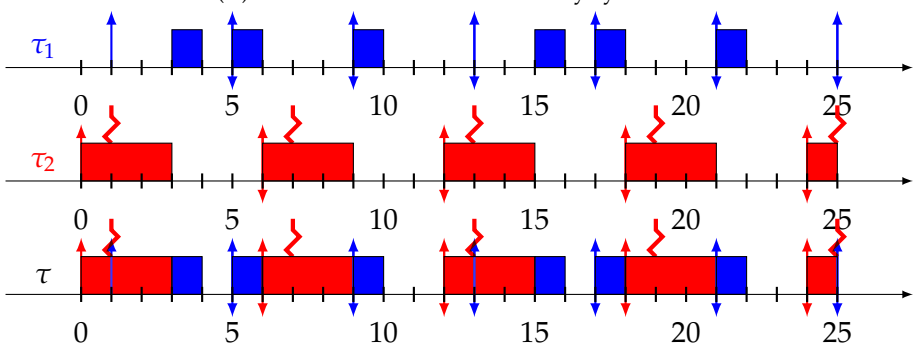
Propriété 4 (Période d'étude pour Dual Priority)
La période d'étude pour Dual Priority dans le scénario synchrone ne s'arrête pas nécessairement à la première période d'inactivité du processeur (c'est-à-dire au premier moment où aucune instance de tâche n'est à exécuter), contrairement à RM ([110]) ou EDF ([152]).

Démonstration. Nous prouvons ces trois propriétés à travers un exemple donné par la Figure 6.5. Nous montrons deux sous-figures dans lesquelles le même jeu de tâches, à savoir $\tau_1(1,4,3,1,4)$ et $\tau_2(3,6,4,2,1)$, est exécuté une première fois sans offset (Figure 6.5a) et une deuxième fois avec un offset de 1 pour τ_1 (Figure 6.5b). Pour les Propriétés 2 et 3, nous n'utilisons que la Figure 6.5a. En effet, nous montrons dans cette figure que le temps de réponse de la tâche τ_1 est de 1 pour la première et la seconde instance de tâche. En revanche, le temps de réponse passe à 2 pour la troisième instance de tâche. Ainsi, dans un scénario synchrone, le temps de réponse de la première instance de tâche n'est pas nécessairement le pire, donc, les candidats d'instant critique ne sont pas connus. De plus, nous montrons dans la Figure 6.5b que dans un scénario non synchrone, avec offset, le pire temps de réponse de τ_1 est de 3. Un scénario synchrone ne permet donc ni de déterminer un candidat d'instant critique ni de déterminer le temps de réponse pire cas d'une tâche.

Pour la Propriété 4, nous montrons dans la Figure 6.5a qu'à l'instant 5 aucune instance de tâche n'est exécutée. Or à cet instant, seules la première et la seconde instance de tâches ont été exécutées pour τ_1, alors que son pire temps de réponse est donné par la troisième instance de tâche. La période d'étude du système n'est donc pas donnée par le premier instant d'inactivité du processeur. □



(A) Ordonnancement Dual Priority synchrone



(B) Ordonnancement Dual Priority avec offset de 1 sur τ_1

FIGURE 6.5 : Ordonnancement du jeu de tâches composé de $\tau_1(1,4,3,1,4)$ et $\tau_2(3,6,4,2,1)$ en Dual Priority

À travers ces différentes propriétés, nous pouvons finalement déterminer que Dual Priority est un ordonnancement quasiment optimal, ce qui permet donc d'ordonnancer un plus grand nombre de jeux de tâches qu'avec un ordonnancement où les tâches ont des priorités fixes. En revanche, l'analyse d'un tel ordonnancement est compliquée et se fera le plus souvent par simulation, et le choix des paramètres des tâches pour appliquer Dual Priority peut être chronophage et peu de méthodes sont proposées. C'est pour répondre à ces derniers problèmes que nous étudions par la suite différents algorithmes permettant de déterminer les paramètres des tâches pour Dual Priority, que ce soit les deux priorités des tâches ou leur échéance intermédiaire.

6.2 Algorithme FDMS avec RM/RM pour les priorités

Dans cette section, nous présentons maintenant un algorithme pour déterminer les échéances intermédiaires des tâches pour l'ordonnancement Dual Priority. Les priorités des tâches suivent l'ordre RM pour les deux bandes de priorités distinctes (toutes les priorités après promotion sont supérieures aux priorités avant promotion). Nous notons cet assignement de priorités Rate-Monotonic/Rate-Monotonic (RM/RM). Cet algorithme ne permettant de trouver que les échéances arbitraires, il serait possible de l'utiliser avec d'autres assignements de priorités, mais nous conserverons RM/RM, celui-ci étant conjecturé au moment de l'étude comme optimal (et seulement quasi-optimal à ce jour). Les jeux de tâches sur lesquels les priorités RM/RM et l'algorithme FDMS ne sont pas ordonnançables étant extrêmement rares, cette approche sera appliquée.

Comme expliqué dans la Section 6.1, cet algorithme fonctionne par simulations. Une simulation doit donc être effectuée sur une hyper-période (cas synchrone), ce qui selon les jeux de tâches peut être vite chronophage si l'hyper-période est élevée. L'algorithme se compose de deux sous algorithmes : Algorithme 13 et 14.

Pour résumer les deux algorithmes, le premier, Algorithme 13 boucle jusqu'à devoir s'arrêter en testant un jeu de tâches avec des échéances intermédiaires actualisées. Ainsi, il s'arrête si les tâches n'ont pas pu avoir d'échéances intermédiaires ou si le jeu de tâches est ordonnançable. Le second Algorithme 14, quant à lui, actualise toutes les valeurs des échéances arbitraires, si besoin, selon les résultats de la simulation faite sur ce jeu de tâches.

Algorithme 13 : FDMS

Entrées : τ
Sorties : τ

```
1 booléen continuerSimu;
2 (continuerSimu, τ) = CalculEchanceIntermediaire(τ);
3 tant que continuerSimu = vrai et τ != ∅ faire
4 |  (continuerSimu, τ) = CalculEchanceIntermediaire(τ);
5 si τ = ∅ alors
6 |  retourner ∅;
7 sinon
8 |  retourner τ;
```

Commençons par décrire l'Algorithme 13. Cet algorithme ne prend qu'une entrée et qu'une seule sortie : le jeu de tâches. Plus précisément, il prend en entrée un jeu de tâches Dual Priority, avec donc les deux priorités associées, qui seront ici basées sur RM/RM. De plus, les échéances intermédiaires sont ici égales à la période, ainsi $S_i = T_i$ pour toute tâche τ_i du jeu de tâches τ. La sortie de l'algorithme est ensuite ce même jeu de tâches, avec les S_i actualisés pour que les tâches respectent leurs échéances. Si les S_i n'ont pas pu être trouvés, l'algorithme retourne à la place un jeu de tâches vide ($\emptyset$), qui indique qu'aucune attribution des échéances intermédiaires n'a pu être trouvée.

La Ligne 1 initialise une variable booléenne qui permettra de déterminer si une réponse pour l'ordonnancement a été trouvée ou si les simulations doivent continuer. La Ligne 2 permet d'initialiser une première valeur des échéances intermédiaires trouvées par la deuxième partie de l'Algorithme 14. Cet algorithme renvoie deux variables : le booléen permettant de déterminer s'il faut continuer ou non les simulations et le jeu de tâches avec de nouvelles valeurs pour les S_i. Si aucune valeur ne peut être trouvée pour les S_i, cet algorithme renvoie alors un jeu de tâches vide ($\emptyset$) au lieu du jeu de tâches.

C'est ce que nous utiliserons pour déterminer si les simulations doivent être continuées ou arrêtées. Ainsi, la boucle Ligne 3 teste si les simulations doivent être continuées et si le jeu de tâches a des paramètres valables. Si tel est le cas, nous relançons alors une simulation Ligne 4 avec de nouvelles valeurs pour la variable booléenne et le jeu de tâches.

Nous testons ensuite la valeur donnée au jeu de tâches : si celle-ci vaut $\emptyset$ (Ligne 5), alors aucune valeur des S_i ne peut être trouvée et l'algorithme renvoie alors un jeu de tâches vide ($\emptyset$) (Ligne 6). En revanche, si l'algorithme a trouvé un résultat, celui-ci est alors renvoyé Ligne 8.

Algorithme 14 : CalculEchanceIntermediaire

Entrées : τ
Sorties : (booléen continuerSimu, τ)

```
1 τ_i = simulationDual(τ);
2 si τ_i = FIN alors
3     retourner (faux, τ);
4 si S_i = 0 alors
5     retourner (faux, ∅);
6 S_i = S_i − 1;
7 retourner (vrai, τ);
```

Nous passons maintenant à l'Algorithme 14 appelé aux Lignes 2 et 4. Cet algorithme prend en entrée le jeu de tâches actualisé pour les valeurs des S_i. Celui-ci devra ensuite les actualiser avant de les retourner. Les sorties sont donc tout d'abord la variable booléenne pour continuer ou arrêter les simulations, pour trouver de nouveaux S_i et le jeu de tâches actualisé.

La Ligne 1 commence par effectuer une simulation du jeu de tâches d'après l'algorithme d'ordonnancement Dual Priority. Puisque toutes les tâches ont tous les paramètres définis, celui-ci pourra alors être simulé correctement. La simulation devra s'arrêter dans deux cas :

- Soit à la première échéance ratée d'une instance de tâche.
- Soit à la fin de l'hyper-période du jeu de tâches.

Dans le premier cas, lors d'une échéance ratée, la simulation de Dual Priority doit alors retourner la tâche τ_i sur laquelle la première échéance a été ratée. Dans le second cas, la simulation retourne la valeur FIN, ce qui nous permettra de déterminer qu'aucune échéance n'a été ratée.

Les lignes suivantes de l'algorithme permettent de tester quelles valeurs retourner et d'actualiser le jeu de tâches avec de nouveaux paramètres. Ainsi, la Ligne 2 teste si aucune échéance n'a été ratée.

(i) Si tel est le cas, l'algorithme retourne alors les valeurs $faux$ et le jeu de tâches τ (Ligne 3). Cela permettra à l'Algorithme 13 de s'arrêter et de retourner le jeu de tâches τ avec des valeurs où toutes les échéances sont respectées.

(ii) Si une échéance a été ratée, la Ligne 4 teste si l'échéance intermédiaire de la tâche, associée à la première instance de tâche qui rate son échéance, peut être soustraite de 1. Dans le cas particulier où celle-ci est déjà égale à 0, le jeu de tâches ne pourra pas avoir de nouvelles valeurs possibles pour les S_i. L'algorithme retourne alors les valeurs $faux$ et $\emptyset$ (Ligne 5). Cela permet donc à l'Algorithme 13 de savoir qu'il peut s'arrêter, mais également qu'aucune échéance intermédiaire ne peut être trouvée pour la tâche considérée, et donc que le jeu de tâches n'est pas ordonnançable.

(iii) Si aucun des deux cas précédents ne se produisent, nous pouvons alors diminuer l'échéance intermédiaire S_i de la tâche τ_i dont l'échéance n'est pas respectée (Ligne 6). La dernière étape est ensuite de retourner les valeurs $vrai$ et τ contenant le même jeu de tâches avec une valeur modifiée pour le S_i de τ_i dont l'échéance n'était pas respectée (Ligne 7). Cela permettra ainsi à l'Algorithme 13 de continuer de chercher de nouvelles échéances intermédiaires et d'avoir des valeurs actualisées pour celles-ci.

Il est facile d'adapter cet algorithme d'ordonnancement à l'ordonnancement de VMs avec notre modèle hiérarchique à deux niveaux. Pour cela, il suffit de connaître l'ordonnancement de la VM et de l'envoyer à l'algorithme. Cela donne les Algorithmes 15 et 16.

Algorithme 15 : FDMS avec prise en compte des VMs

Entrées : τ, ordoVM
Sorties : τ

```
1 booléen continuerSimu;
2 (continuerSimu, τ) = CalculEchanceIntermediaire(τ, ordoVM);
3 tant que continuerSimu = vrai et τ != ∅ faire
4 |  (continuerSimu, τ) = CalculEchanceIntermediaireVM(τ, ordoVM);
5 si τ = ∅ alors
6 |  retourner ∅;
7 sinon
8 |  retourner τ;
```

Algorithme 16 : CalculEchanceIntermediaireVM

Entrées : τ, ordoVM
Sorties : (booléen continuerSimu, τ)

```
1 τ_i = simulationDualVM(τ, ordoVM);
2 si τ_i = FIN alors
3 |  retourner (faux, τ);
4 si S_i = 0 alors
5 |  retourner (faux, ∅);
6 S_i = S_i − 1;
7 retourner (vrai, τ);
```

Dans ces deux algorithmes, les seuls changements sont les entrées, à savoir l'ajout de $ordoVM$, qui contient l'ordonnancement de la VM étudiée, et donc si la VM est ordonnancée ou non à chaque instant. Ces changements sont valables pour les Algorithmes 15 et 16. Puisque l'Algorithme 15 appelle l'Algorithme 16 aux Lignes 2 et 4, les appels doivent donc être changés pour prendre en compte le paramètre $ordoVM$. Dans l'Algorithme 16, le seul changement est fait à la Ligne 1. Ici, ce n'est plus une simulation classique qui doit être effectuée mais une simulation qui prend en compte si la VM étudiée est ordonnancée ou non à l'instant étudié. Par exemple, à l'instant 4, il faut d'abord vérifier si la VM est ordonnancée pour pouvoir exécuter la tâche voulue. Ce sont les seuls changements nécessaires pour adapter l'algorithme FDMS au modèle hiérarchique à deux niveaux.

Nous montrons maintenant à travers un exemple l'intérêt de Dual Priority appliquée à notre modèle. Le jeu de tâches étudié est donné dans la Table 6.5. Ce jeu de tâches est exécuté dans la VM $\rho^1(7, 10)$. L'ordonnancement seul de la VM est donné par la Figure 6.6. Elle est ainsi ordonnancée aux instants 0 à 3, à l'instant 4, 6 et 7 ainsi que 9. Ce motif d'ordonnancement se répète toutes les 10 unités de temps, d'après la période de la VM. Nous montrons également dans la Figure 6.6 que l'ordonnancement de ce jeu de tâches dans cette VM échoue avec l'ordonnancement RM pour la tâche τ_3 : son échéance n'est pas respectée à l'instant 50.

Tâche	C_i	T_i	D_i
τ_1	2	10	10
τ_2	5	20	20
τ_3	11	50	50

TABLE 6.5 : Jeu de tâche qui n'est pas ordonnançable par RM dans la VM $\rho^1(7,10)$

Tâche	C_i	T_i	D_i	P_i^1	P_i^2	S_i
τ_1	2	10	10	4	1	10
τ_2	5	20	20	5	2	20
τ_3	11	50	50	6	3	49

TABLE 6.6 : Jeu de tâche Dual Priority devenu ordonnannçable dans la VM $\rho^1(7,10)$

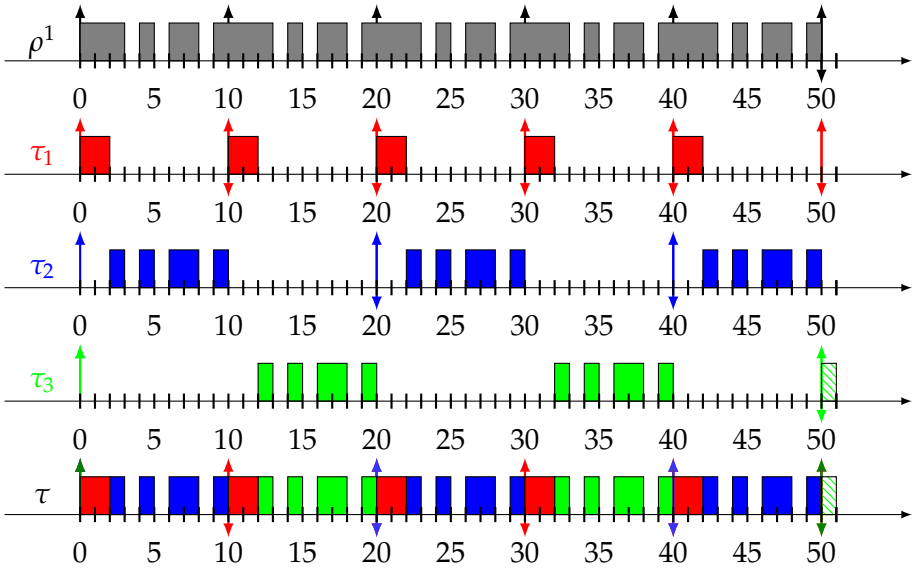


FIGURE 6.6 : Ordonnancement du jeu de tâches de la Table 6.5 dans la VM $\rho^1(7,10)$

D'après l'Algorithme 15, il faut donc réduire l'échéance intermédiaire de τ_3 d'une unité. Nous obtenons ainsi le nouveau jeu de tâches donné dans la Table 6.6. Nous montrons par la Figure 6.7, qu'avec ce seul changement dans les échéances intermédiaires, le jeu de tâches qui n'était alors pas ordonnançable avec RM devient ordonnançable avec Dual Priority.

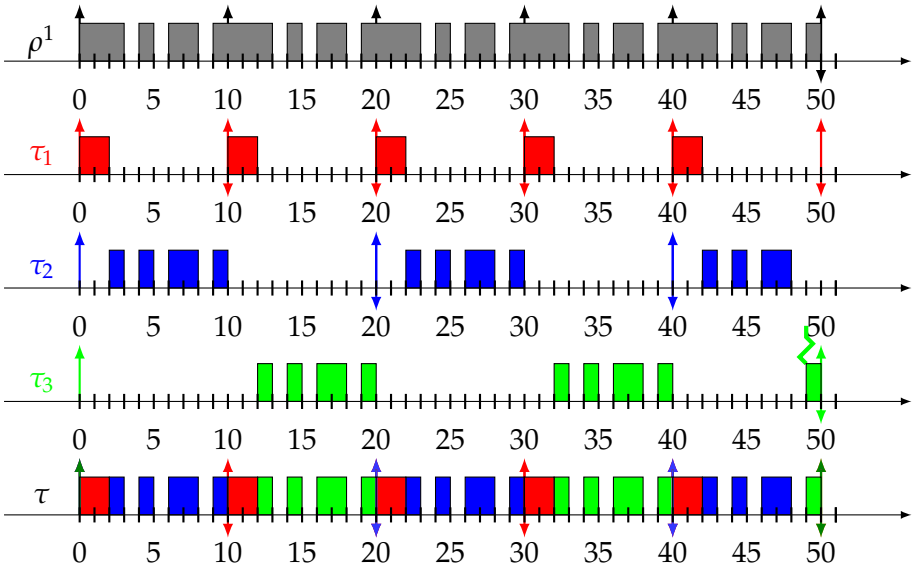


FIGURE 6.7 : Ordonnancement du jeu de tâches de la Table 6.6 dans la VM $\rho^1(7, 10)$

Dual Priority, bien que longtemps conjecturée optimale, avec son taux de succès très élevé, a été montrée non optimale dans l'état de l'art (cf. Section 6.1). Le principal inconvénient de cet algorithme est le temps de calcul des échéances intermédiaires. En effet, leur calcul peut demander un grand nombre de simulations : jusqu'au produit de toutes les périodes dans le pire cas (en considérant que les échéances intermédiaires peuvent prendre en pire cas toutes les valeurs possibles entre 0 et la période des tâches). Selon le jeu de tâches de base et l'hyper-période, cela peut conduire à un nombre de simulations exponentiel (non réalisable). C'est pour cela que nous voyons par la suite une manière bien plus simple et rapide d'assigner les échéances intermédiaires.

6.3 Algorithme RM^{-1}/RM avec RML pour les priorités

Dans cette section, nous présentons une manière de déterminer rapidement les échéances intermédiaires. L'algorithme présenté pour déterminer ces échéances est appelé Rate Monotonic Laxity (RML). Les priorités sont assignées par RM^{-1}/RM (en deux bandes distinctes).

Définition 56 (RM^{-1})
En utilisant l'algorithme RM^{-1}, les priorités sont assignées à l'inverse de RM. Ainsi, plus la période est grande, plus la priorité est haute. La tâche avec la plus petite période a donc cette fois la priorité la plus basse.

Dans cet algorithme, il existe deux types de tâches : les tâches dites Lowest Priority Viable (LPV) et les tâches qui seront prises en compte par l'algorithme RM^{-1}/RM. Une tâche est dite LPV si et seulement si toutes ses instances de tâches respectent leurs échéances quand :

- Leur priorité est attribuée par l'algorithme d'Audsley (voir Chapitre 2), sans promotion, en partant de la plus basse priorité. Dans le cas particulier de tâches à échéance sur requête, nous considérons l'ordonnancement RM (optimal dans ce contexte). Ainsi, la première tâche LPV a la plus petite priorité, la seconde la deuxième plus basse priorité, et ainsi de suite.

Tâche	C_i	T_i	D_i	P_i^1	P_i^2	S_i	LPV
τ_1	3	6	6	4	1	3	Non
τ_2	4	9	9	5	2	0	Non
τ_3	2	36	36	6	3	0	Oui

TABLE 6.7 : Jeu de tâche Dual Priority pour démontrer l'importance d'une tâche LPV

- Les autres tâches ont toutes une priorité plus importante (peu importe la priorité).
- Lors de l'ordonnancement des tâches non LPV, l'ordonnanceur considère des échéances souples, c'est-à-dire que si une échéance est dépassée, le système continue de l'ordonnancer jusqu'à ce qu'elle termine son exécution.

Nous montrons l'importance de donner une priorité moins importante à ces tâches et donc de les exécuter en tâches de fond grâce au jeu de tâches donné dans la Table 6.7. Ce jeu de tâches est donc constitué de trois tâches distinctes. Parmi ces tâches, seule la troisième est une tâche LPV.

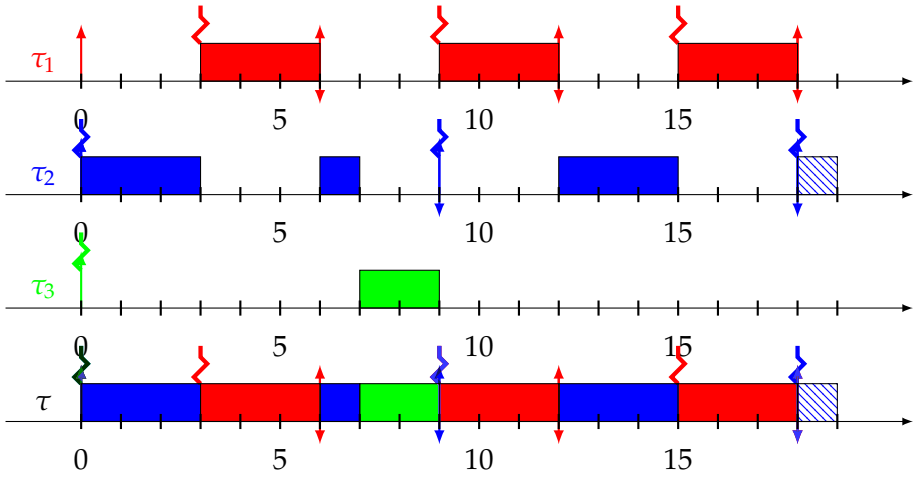


FIGURE 6.8 : Ordonnancement du jeu de tâches donné dans la Table 6.7 sans considérer l'ordonnancement LPV

Nous montrons, dans un premier temps, dans la Figure 6.8, l'ordonnancement du jeu de tâches précédemment défini en utilisant Dual Priority, sans prendre en compte les propriétés LPV potentielles des tâches. Cet ordonnancement échoue alors à l'instant 18 lorsque la tâche τ_2 dépasse son échéance du fait de l'ordonnancement de la tâche τ_3.

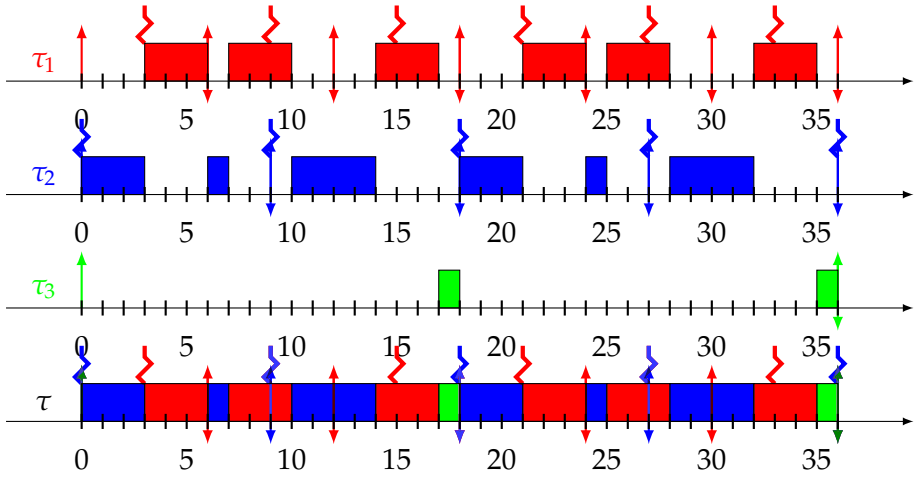


FIGURE 6.9 : Ordonnancement du jeu de tâches donné dans la Table 6.7 en prenant en compte l'ordonnancement LPV

Nous montrons ensuite dans la Figure 6.9 l'ordonnancement du même jeu de tâches en utilisant cette fois la propriété LPV sur la tâche τ_3. Elle est donc ordonnancée avec une priorité plus basse que les deux autres, et son échéance intermédiaire n'est alors plus prise en compte. Avec un tel jeu de priorités, la tâche τ_3 va cette fois s'exécuter aux deux instants où aucune des deux autres tâches ne sont exécutées, à savoir aux instants 17 et 35. Ainsi, toutes les tâches du système respectent leurs échéances, contrairement à précédemment quand les propriétés LPV des tâches n'étaient pas utilisées.

En utilisant l'assignement de priorités RM^{-1}/RM, les tâches dites LPV sont ordonnancées dans une bande de priorité inférieure à toutes les autres tâches. En effet, ces tâches n'auront qu'une seule priorité, qui sera définie à l'aide de l'algorithme d'Audsley (cf chapitre 2.1). L'algorithme pour déterminer les priorités des tâches LPV est présenté dans l'Algorithme 17 qui est une application d'Audsley dans le cas où des tâches sont ordonnancées grâce à Dual Priority

Cet algorithme prend en entrée le jeu de tâches complet τ et prend en sortie un sous-jeu de tâches τ_{lpv} correspondant aux tâches qui peuvent être traitées comme étant LPV et pour lesquelles les priorités sont assignées par cet algorithme, ainsi que la liste des priorités associées à ces dernières tâches. Le jeu de tâches en entrée τ est réduit pour chaque tâche qui sera traitée comme LPV. Ces tâches particulières passeront dans le sous jeu de tâches τ_{lpv}.

Les deux premières lignes permettent d'initialiser des variables pour l'algorithme : la Ligne 1 initialise une variable booléenne qui permet de déterminer quand arrêter l'algorithme et la Ligne 2 initialise un ensemble vide pour les tâches qui seront LPV.

La Ligne 3 est le début d'une boucle jusqu'à la Ligne 10 pour déterminer les tâches qui peuvent être LPV. La Ligne 4 réinitialise la variable booléenne à chaque début de boucle pour que la suite détermine si la boucle doit être arrêtée.

Une seconde boucle, Ligne 5, permet de déterminer les tâches qui sont LPV au sein du jeu de tâches τ. Si aucune tâche n'est trouvée comme étant LPV, la boucle Ligne 3 est alors arrêtée.

La Ligne 6 teste si une tâche est LPV. Pour ce faire, il existe plusieurs méthodes : un calcul de temps de réponse pire cas, une simulation ou d'autres conditions nécessaires et suffisantes. Ce calcul doit se faire avec le jeu de tâches τ qui peut contenir moins de tâches que le jeu de tâches initial. En effet, comme l'indique la Ligne 7,

Algorithme 17 : Priorités des tâches LPV

Entrées : τ
Sorties : τ_{lpv}, *priorités*

```
1  booléen continuer;
2  τ_lpv = ∅;
3  répéter
4      continuer = faux;
5      pour chaque τ_i ∈ τ faire
6          si τ_i est LPV alors
7              τ = τ \ τ_i;
8              τ_lpv = τ_lpv ∪ τ_i;
9              continuer = vrai;
10 jusqu'à continuer = faux;
11 n = | τ |;
12 j = | τ_lpv |;
13 priorités = liste();
14 pour i = 1; i ≤ j; i = i + 1 faire
15     priorités+ = (2 × n + j − i + 1);
16 retourner (τ_lpv, priorités);
```

chaque tâche considérée comme LPV est supprimée du jeu de tâches initial et ajoutée à l'ensemble des tâches LPV (Ligne 8). Si au moins une tâche du jeu de tâches τ actualisé a été trouvée comme étant LPV, la Ligne 9 permet d'effectuer un nouveau test avec le jeu de tâches τ contenant moins de tâches.

Une fois toutes les tâches LPV déterminées, l'objectif suivant est de fixer les priorités de ces tâches en fonction du nombre de tâches non LPV. Pour ce faire, des variables doivent être initialisées. Il s'agit :

- Du nombre de tâches contenues dans τ, enregistré dans la variable n (Ligne 11), après que les tâches LPV aient été retirées de cet ensemble.
- Du nombre de tâches contenues dans τ_{lpv}, enregistré dans la variable j (Ligne 12).
- De la liste des priorités associées aux tâches LPV (Ligne 13).

La ligne suivante (Ligne 14) parcourt toutes les tâches de l'ensemble des tâches LPV. Pour toutes ces tâches, la priorité est ensuite affectée Ligne 15 par la formule $(2 \times n + j - i + 1)$. En effet, cette formule correspond à deux fois le nombre de tâches non LPV, puisque celles-ci seront de type Dual Priority, il faut donc pouvoir leur allouer chacune deux priorités, soit au total $2 \times n$ priorités. La suite de cette formule est logique : il s'agit de mettre la priorité la plus basse à la première tâche LPV trouvée, la seconde priorité la plus basse à la seconde et ainsi de suite.

La dernière ligne, Ligne 16 permet de retourner l'ensemble des tâches LPV avec leurs priorités associées. La particularité de cet algorithme est qu'il va extraire du jeu de tâches initial τ un sous-ensemble de tâches correspondant aux tâches LPV avec leurs priorités associées. Après son application, le jeu de tâches initial τ est ainsi réduit.

Maintenant que toutes les tâches qui n'ont pas besoin de deux niveaux de priorité sont écartées, nous pouvons revenir sur l'algorithme RM^{-1}/RM avec RML pour l'assignation des priorités.

τ_i	C_i	T_i	D_i	P_i^1	P_i^2	S_i
τ_1 (τ'_1)	3	6	6	6 (4)	1 (1)	3
τ_2 (τ'_2)	4	9	9	5 (3)	2 (2)	0
τ_3 (τ'_3)	2	36	36	4 (5)	3 (5)	0 (36)

TABLE 6.8 : Jeu de tâches τ défini sans considérer de tâches LPV (τ_1, τ_2, τ_3) puis avec la tâche τ'_3 considérée comme LPV (et les tâches τ'_1 et τ'_2 considérées comme non LPV)

Commençons par définir RML. La laxité d'une instance de tâche déterminée par RM est caractérisée ainsi :

- La laxité est la différence entre la date de fin de son exécution et son échéance absolue quand le système est ordonnancé par RM si la tâche est ordonnançable par RM.
- La laxité est égale à 0 si la tâche n'est pas ordonnançable par RM.

Pour calculer cette laxité, une possibilité est offerte : calculer le temps de réponse pire cas à l'aide de la formule adaptée à RM. De plus, si l'échéance d'une instance de tâche est dépassée, le calcul peut être arrêté et l'échéance intermédiaire trouvée est ainsi égale à 0. Cela donne pour une tâche τ_i avec échéance implicite :

$$S_i = max(0, D_i - R_i)$$

Pour rappel, S_i est l'échéance intermédiaire de la tâche τ_i, D_i son échéance et R_i son temps de réponse pire cas. Ce dernier est calculé par :

$$R_i = \min_{t>0} \left\{ t = \sum_{k<i} \left\lceil \frac{t}{T_k} \right\rceil C_k + C_i \right\}$$

Ces formules peuvent être amenées à changer selon le contexte. En effet, si par exemple les échéances ne sont plus implicites ou contraintes, il faudra alors utiliser une autre formule pour calculer le temps de réponse pire cas (voir Chapitre 2). Nous considérons dans ce chapitre le cas d'échéances implicites : la première instance de chaque tâche est à considérer pour obtenir son pire temps de réponse. Grâce à cela, le nombre d'itérations pour le calcul des échéances intermédiaires est borné par $n \times \max_{i=1,\dots,n} T_i$ pour un jeu de tâches à n tâches. En effet, le calcul du point fixe lors du calcul de temps de réponse peut être arrêté aussitôt que l'échéance est dépassée, et puisque $D_i = T_i$, aussitôt que la période est dépassée.

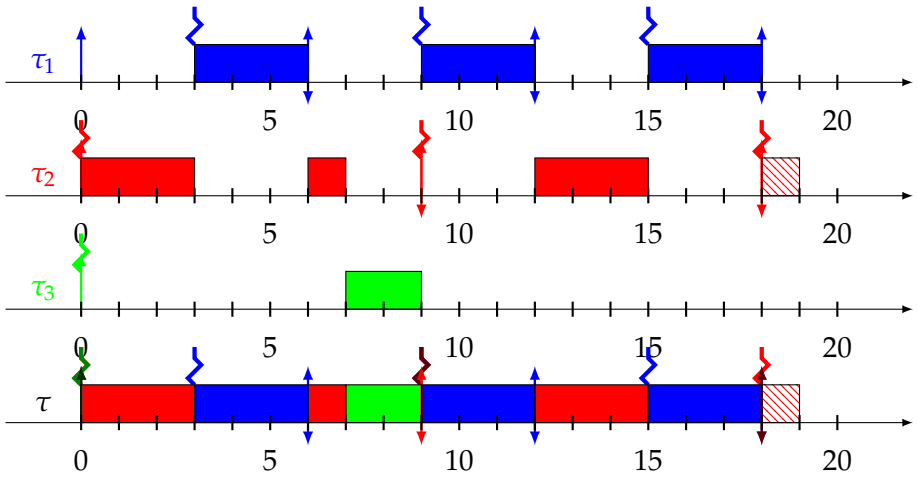
Ainsi, nous avons vu comment affecter les priorités et également comment calculer les échéances intermédiaires. Nous avons donc tous les paramètres pour l'algorithme Dual Priority. Les tâches LPV qui n'ont qu'une seule priorité peuvent être prises en compte par Dual Priority avec les paramètres suivants :

$$\forall \tau_i \in \tau_{lpv}, \tau_i(C_i, T_i, D_i, P_i^1, P_i^2 = P_i^1, D_i)$$

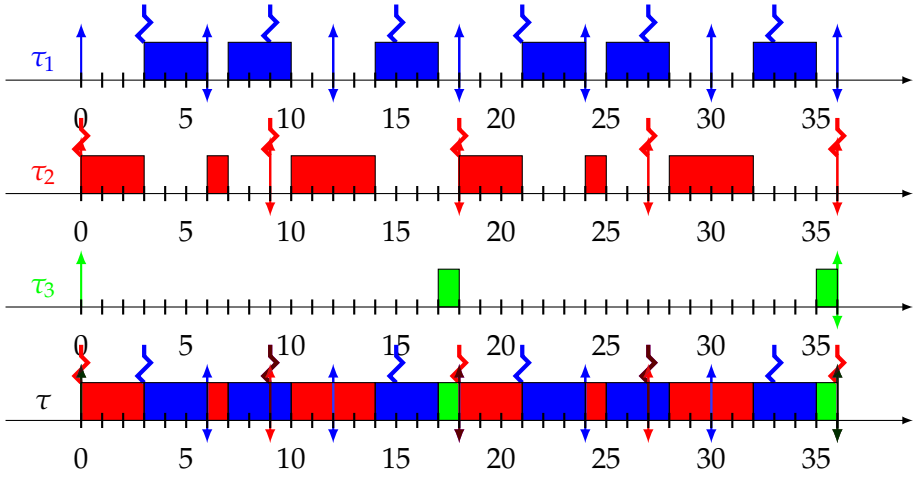
La prise en compte de ces tâches est essentielle au bon fonctionnement de l'algorithme : sur certains jeux de tâches, l'inclusion des tâches LPV empêche de trouver un ordonnancement valide. Celles-ci doivent donc être prises en compte ensuite, avec une seule priorité trouvée par l'Algorithme 17.

Nous exposons à travers la Table 6.8 deux jeux de tâches :

- Un premier jeu de tâches, composé de τ_1, τ_2 et τ_3 sur lequel a été appliqué l'algorithme RM^{-1}/RM avec RML pour fixer les priorités, directement sans prendre en compte les tâches LPV.
- Un second jeu de tâches, composé de τ'_1, τ'_2 et τ'_3 sur lequel les tâches LPV ont été testées par l'Algorithme 17. Seule τ'_3 est LPV. Les paramètres différents par rapport au jeu de tâches précédent sont indiqués entre parenthèses. Ceux qui sont similaires ne sont pas reprécisés.



(A) Jeu de tâches τ_1, τ_2, τ_3 de la Table 6.8 ordonnancé en Dual Priority sans considérer de tâches LPV



(B) Jeu de tâches τ'_1, τ'_2, τ'_3 de la Table 6.8 ordonnancé en Dual Priority en considérant τ'_3 comme tâche LPV

FIGURE 6.10 : Comparaison du jeu de tâches de la Table 6.8 sans considérer de tâches LPV puis en les prenant en compte

Nous montrons à travers la Figure 6.10 l'exemple d'un même jeu de tâches dont les paramètres sont calculés par l'algorithme Dual Priority vu dans cette section, à savoir RM^{-1}/RM avec RML pour les priorités, mais une première fois sans prendre en compte les tâches LPV (Figure 6.10a), puis en considérant les tâches LPV, à savoir τ'_3 dans la Figure 6.10b.

Nous pouvons ainsi observer qu'en ne considérant pas les tâches LPV, les jeux de tâches ne sont pas tous ordonnançables grâce à l'approche RM^{-1}/RM avec RML pour l'attribution des priorités. En revanche, en n'utilisant cette approche que pour les tâches non LPV et en utilisant l'Algorithme 17 pour les tâches LPV, qui permet de

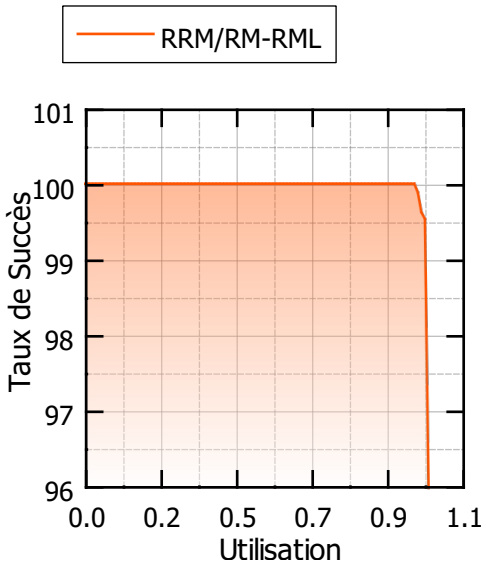


FIGURE 6.11 : Taux de succès de l'approche RM^{-1}/RM avec RML pour les priorités en fonction de l'utilisation

les ordonnancer en "tâche de fond", à savoir avec une priorité plus basse que toutes les tâches Dual Priority à proprement parler, cela permet d'ordonnancer toutes les tâches.

Tout l'intérêt de cette approche par rapport à FDMS est que celle-ci calcule tous les paramètres pour Dual Priority de façon simple et rapide, contrairement à FDMS qui trouvait les paramètres de manière itérative en testant un grand nombre de simulations. Une autre différence principale est que FDMS avait un taux de succès proche de 100%, alors que cette approche a un taux de succès moins élevé mais meilleur que pour l'ordonnancement RM.

Nous montrons ainsi dans la Figure 6.11 le taux de succès de RM^{-1}/RM avec RML pour l'assignation des priorités. Ce taux de succès n'est pas de 100%, Dual Priority n'étant pas optimal. Mais comme montré, RM^{-1}/RM avec RML pour l'assignation des priorités, il reste très élevé et ne tombe qu'à 96% lorsque l'utilisation est égale à 1. Concernant la génération des jeux de tâches, nous avons considéré pour la Figure 6.11 77 000 jeux de tâches générés à l'aide de l'algorithme UUnifast ([55]) avec des périodes entre 100 et 100000 et une hyper-période limitée selon l'approche définie dans [75]. Pour que les simulations ne prennent pas trop de temps pour obtenir le résultat, les hyper-périoeont limitées à 999999. Au-delà, les jeux de tâches ne sont pas pris en compte.

En analysant la Figure 6.11, nous pouvons également mettre en évidence que le taux de succès avec une utilisation de 0.98 est égal à 99.6%, ce qui fait de cette solution, un algorithme très efficace avec un grand taux de succès.

En plus du taux de succès pour cette approche, nous avons également relevé le taux de succès de cette approche en ne prenant en compte que les tâches LPV. Ainsi, si toutes les tâches du jeu de tâches ne sont pas considérées comme LPV, l'ordonnancement de ce système est considéré comme ayant échoué.

Nous montrons ce résultat à travers la Figure 6.12. Si toutes les tâches sont considérées comme LPV, le système est alors ordonnançable par RM seul. Cela permet de mettre en évidence que les jeux de tâches ordonnancés par notre approche ont un meilleur taux de succès que ceux ordonnancés par RM. Cela met également en évidence que les jeux de tâches générés ne sont pas biaisés, RM ayant un moins bon

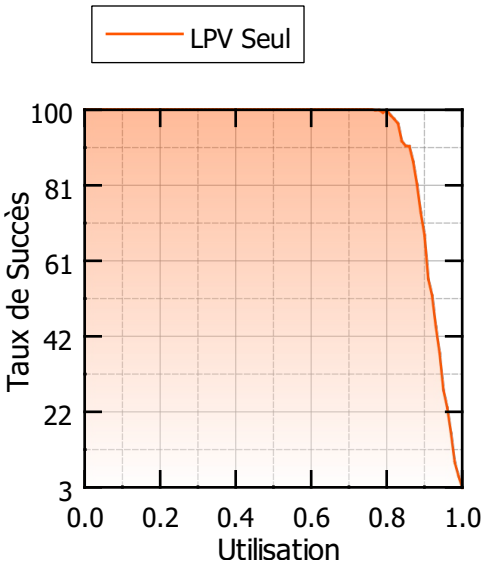


FIGURE 6.12 : Taux de succès de jeux de tâches où toutes les tâches sont LPV

taux de succès que notre approche.

Ainsi, comme montré dans la Figure 6.12, le taux de succès de RM seul est de 100% jusqu'à une utilisation de 0.8 et descend rapidement jusqu'à 3% avec une utilisation proche de 1.

Grâce à cette nouvelle approche, à savoir RM^{-1}/RM avec RML pour l'assignation des priorités, nous avons ainsi une solution pour trouver les paramètres pour des tâches Dual Priority de façon bien plus rapide que ceux trouvés par FDMS, et avec un taux de succès presque aussi bon.

Maintenant que nous savons comment Dual Priority se comporte et quels sont les différents algorithmes, nous pouvons nous intéresser à leur intégration dans notre modèle hiérarchique à deux niveaux, et surtout nous intéresser à ce que cela peut nous apporter.

6.4 Gains obtenus avec Dual Priority

Dans cette section, nous étudions comment intégrer l'ordonnancement Dual Priority au sein de notre modèle hiérarchique à deux niveaux. Le but est ici de réussir à ordonnancer des jeux de tâches précédemment non ordonnançables sans Dual Priority, dans le cas particulier où les tâches ont une activation synchrone (pire cas dans notre contexte de tâches sporadiques). Nous cherchons ainsi à augmenter le taux du succès des différentes approches en les complétant par un ordonnancement Dual Priority si besoin.

Nous avons ainsi créé une nouvelle procédure pour avoir le meilleur taux de succès possible :

- Tester d'ordonnancer les VMs de manière harmonique comme précédemment, à l'aide des Algorithmes 2, 3, 4 et 5.
- Pour toutes les VMs pour lesquelles toutes les tâches ne respectent pas leurs échéances avec la méthode harmonique, on teste alors l'ordonnancement Dual Priority dans le cas particulier où les tâches sont synchones.

- Pour toutes les VMs qui sont restées sans paramètres ou dont toutes les tâches ne respectent pas les échéances, nous trouvons les paramètres de ces VMs à l'aide de l'algorithme proportionnel itératif issu de l'ordonnancement Pfair (Algorithmes 9, 10 et 11).
- L'ordonnancement des tâches pour les VMs ordonnancées par l'algorithme proportionnel itératif est ensuite testé à l'aide de RM.
- Pour chaque VM dont l'ordonnancement ne passe pas avec RM, Dual Priority est testée lorque les tâches sont synchrones.

Dual Priority est ainsi utilisée comme alternative à RM lorsque RM n'a pas pu aboutir à un résultat satisfaisant. Cela permet ainsi d'avoir le plus de systèmes possibles sans Dual Priority pour deux raisons :

- Vérifier le respect des échéances rapidement, Dual Priority nécessitant une simulation jusqu'à l'hyper-période dans le cas de tâches périodiques synchrones.
- Éviter de calculer les paramètres des tâches en Dual Priority, qui peut être très coûteux en temps de calcul.

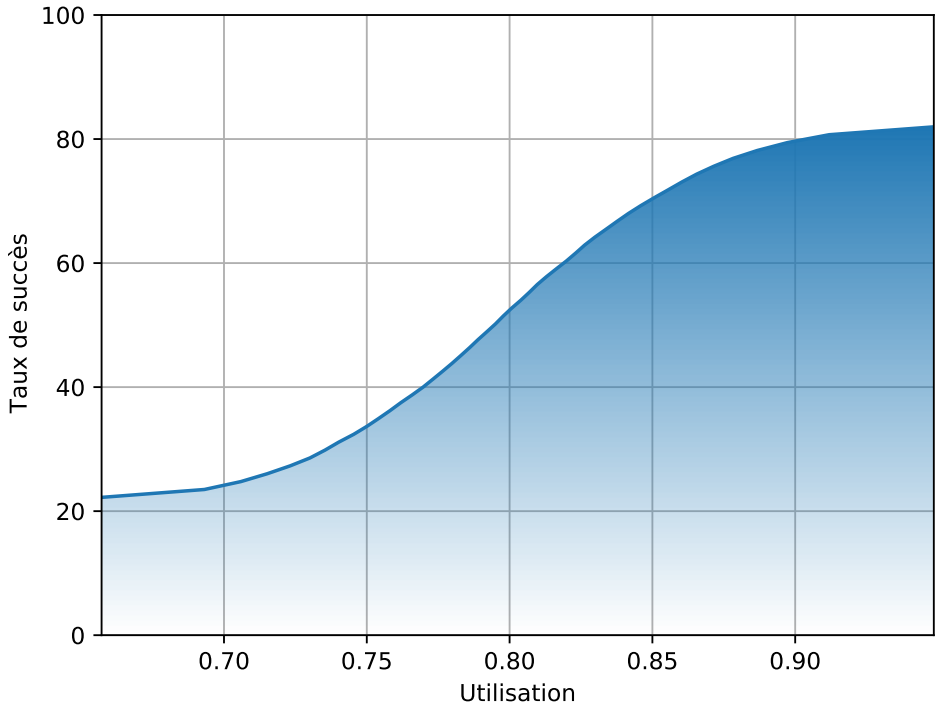
Concernant Dual Priority, l'algorithme ici utilisé est FDMS (Algorithmes 13 et 14). En effet, cet algorithme permet de s'adapter à n'importe quel modèle, y compris un modèle hiérarchique à deux niveaux comme ici présent. Cette propriété vient du fait que l'algorithme teste toutes les possibilités d'échéances intermédiaires tant que le système n'est pas ordonnançable ou qu'une échéance intermédiaire devient négative (système non ordonnançable).

Nous proposons maintenant de comparer les résultats précédents après y avoir ajouté Dual Priority. Le contexte des simulations est le même que celui décrit dans les Sections 4.4 et 5.4. Plus de détails concernant la génération des tâches sont donnés dans la Section 7.1. La section 7.2 du Chapitre 7 détaille l'algorithme utilisé pour effectuer les simulations.

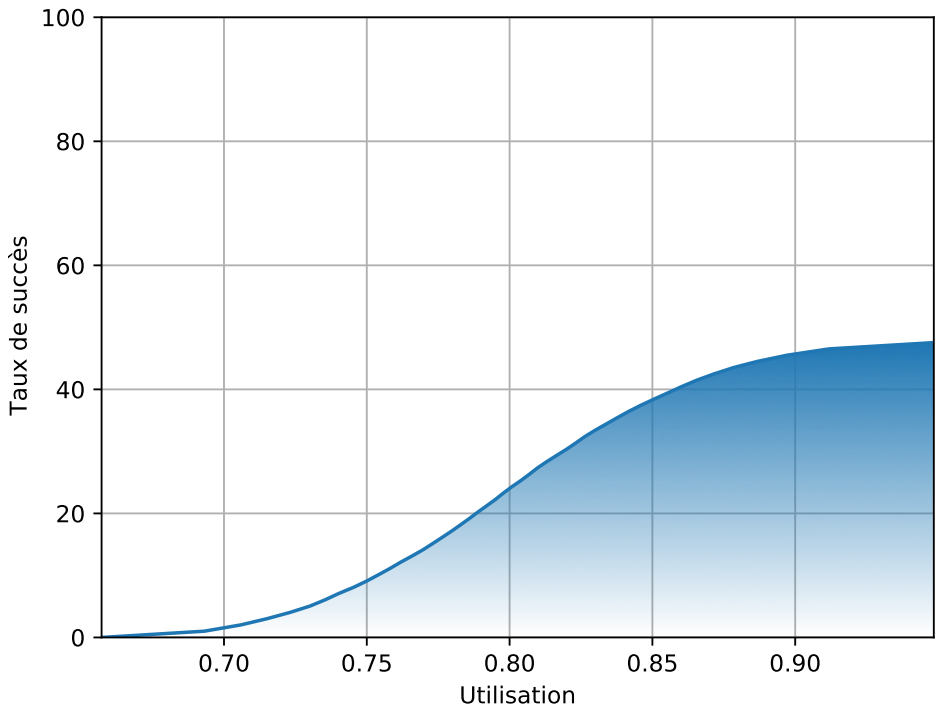
Nous avons ainsi des systèmes à quatre VMs et des utilisations fixées entre 0.1 et 0.2 pour les trois premières VMs et entre 0.3 et 0.4 pour la dernière. Nous commençons par comparer les résultats obtenus dans la Section 4.4 avec l'algorithme harmonique seul. Pour une seule VM, les résultats sont similaires et ne seront donc pas présentés, tout comme pour les résultats sur le taux de succès où exactement quatre VMs sont ordonnancées par l'algorithme harmonique seul. Les résultats sans overheads sont présentés dans les Figures 6.13 et 6.14. D'autres résultats avec overheads sont présentés dans la Figure 6.16.

Dans la Figure 6.13, nous observons que la courbe du taux de succès pour exactement deux VMs ordonnancées en utilisant Dual Priority suit celle de RM utilisé pour l'ordonnancement des tâches. Mais celle-ci est légèrement inférieure : au lieu de commencer avec un taux de succès de 22%, le taux de succès de Dual Priority débute à 0%. Ce taux de succès croît par la suite jusqu'à environ 45% de réussite pour Dual Priority, contrairement à un taux de succès de plus de 80% en priorité fixe. Ce taux est donc légèrement inférieur. Nous l'expliquons à l'aide de la Figure 6.14.

Dans la Figure 6.14, nous observons un taux de succès décroissant allant de 100% pour une utilisation de 0.63 jusqu'à environ 0% avec une utilisation proche de 1. Ce taux de succès part d'une base bien plus haute avec Dual Priority (100% contre 72%) et décroît jusqu'à 0%. Nous avons donc bien plus de systèmes avec trois VMs dont les tâches respectent leurs échéances avec Dual Priority par rapport à un ordonnancement à priorité fixe. C'est ce qui explique la réduction du nombre de systèmes dont

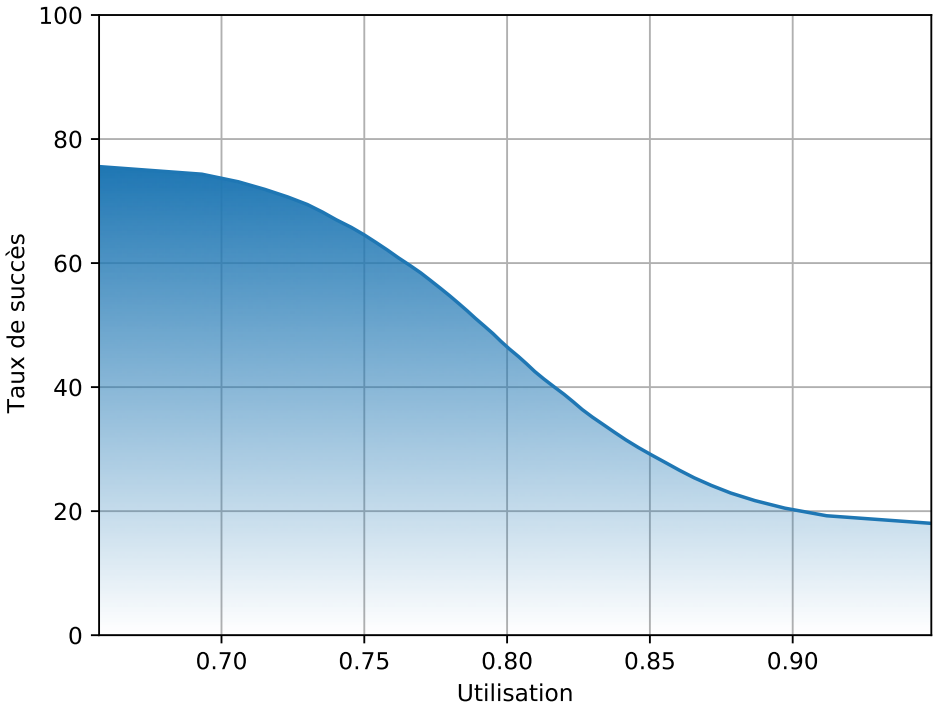


(A) Taux de succès d'exactement deux VMs en fonction de l'utilisation avec RM au niveau des tâches

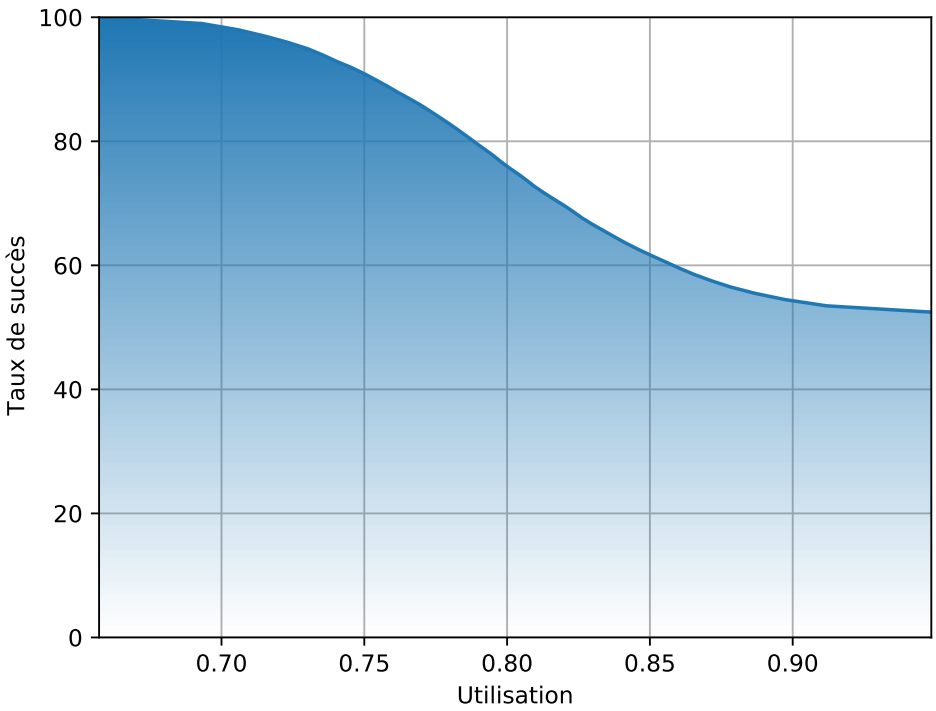


(B) Taux de succès d'exactement deux VMs en fonction de l'utilisation avec DP au niveau des tâches

FIGURE 6.13 : Comparaison entre un ordonnancement à priorité fixe et Dual Priority pour le second niveau et l'algorithme harmonique seul pour le premier niveau où exactement deux VMs sont ordonnancées.



(A) Taux de succès d'exactement trois VMs en fonction de l'utilisation avec FP au niveau des tâches



(B) Taux de succès d'exactement trois VMs en fonction de l'utilisation avec DP au niveau des tâches

FIGURE 6.14 : Comparaison entre un ordonnancement à priorité fixe et Dual Priority pour le second niveau et l'algorithme harmonique seul pour le premier niveau où exactement trois VMs sont ordonnancées.

les tâches respectent leurs échéances avec Dual Priority dans le cas où exactement deux VMs sont ordonnançables et pas plus.

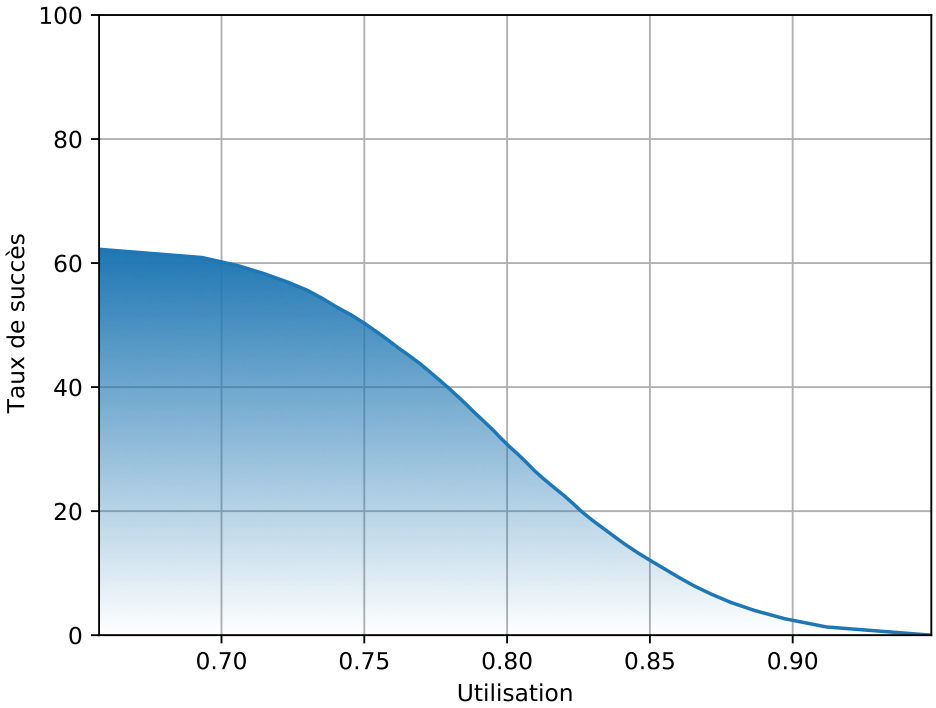
Pour conclure, nous décrivons un dernier taux de succès : celui présenté dans la Section 5.4, en combinant donc l'algorithme proportionnel itératif à l'algorithme harmonique, mais cette fois-ci en utilisant Dual Priority pour l'ordonnancement des tâches. Ce taux de succès est indiqué dans la Figure 6.15.

Dans la Figure 6.15, le taux de succès avec Dual Priority au deuxième niveau commence à presque 90% pour une utilisation de 0.63 et décroît jusqu'à environ 0%, avec une utilisation proche de 1. Cette figure nous permet ainsi de conclure que Dual Priority apporte de réels résultats par rapport à un ordonnancement à priorité fixe au second niveau. Nous passons ainsi d'un taux de succès de 60% avec une utilisation de 0.63 à presque 90%. Le delta étant de 23% pour ce taux d'utilisation, le gain n'est pas négligeable.

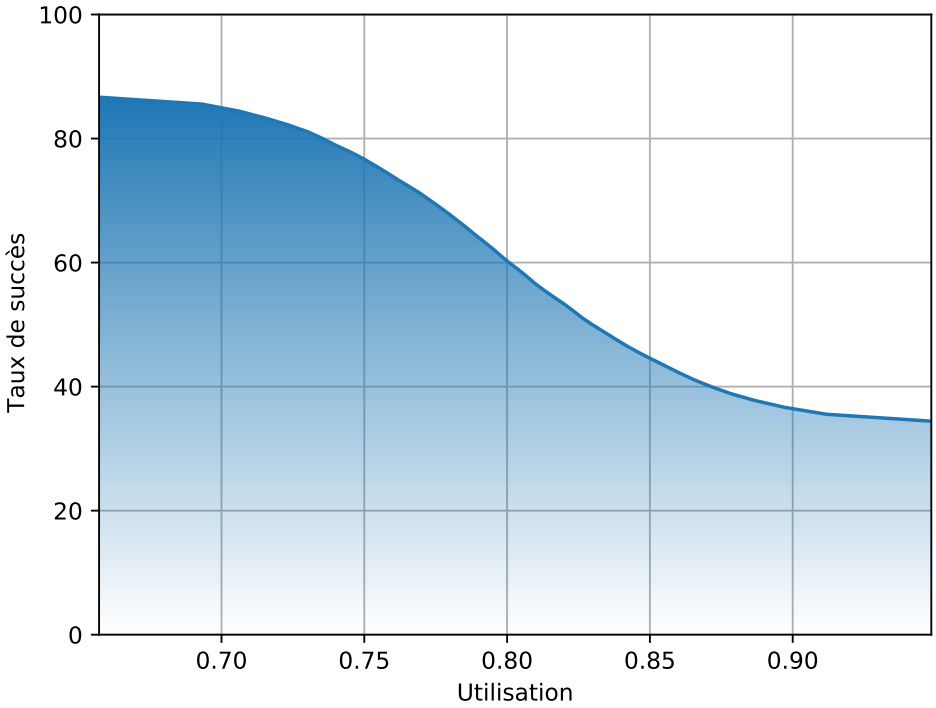
Pour les résultats avec overhead, nous ne présentons que la Figure 6.16. En effet, nous avions vu précédemment dans le Chapitre 4, Section 4.4 qu'avec overhead, le système était ainsi beaucoup plus chargé. Avec de telles utilisations et autant de VMs, très peu de VMs étaient alors ordonnançables. Ainsi, nous ne présentons ici que le cas de l'algorithme harmonique où exactement deux VMs sont ordonnancées.

Nous obtenons une courbe décroissante partant d'un taux de succès d'un peu plus de 3% et descendant jusqu'à 0% pour une utilisation allant de 0.63 à une utilisation proche de 1. Ce taux de succès était précédemment croissant : plus le système était chargé, et plus le nombre de systèmes à uniquement deux VMs dont les tâches respectaient leurs échéances, augmentait. Cette fois, la courbe se rapproche de la courbe d'un même système à trois VMs sans overhead.

Nous décrirons dans le Chapitre 7 l'algorithme complet pour les simulations avec Dual Priority. Nous verrons également que nous n'utilisons que l'algorithme FDMS pour Dual Priority dans ce contexte. Cela vient principalement du fait que cet algorithme reste l'algorithme Dual Priority qui ordonnance le plus de jeux de tâches dans nos simulations malgré sa non optimalité (voir Section 6.2). De plus, dans le contexte d'ordonnancement hiérarchique à deux niveaux, il n'est pas possible d'appliquer Dual Priority avec des priorités assignées par RM^{-1}/RM et les échéances intermédiaires par RML. En effet, le calcul de l'échéance intermédiaire ne prend pas en compte le fait que la VM dans laquelle les tâches s'exécutent n'est pas tout le temps active. Il faudrait pouvoir ajouter ce comportement dans le calcul de l'échéance intermédiaire, ce qui est laissé comme perspective pour la thèse.

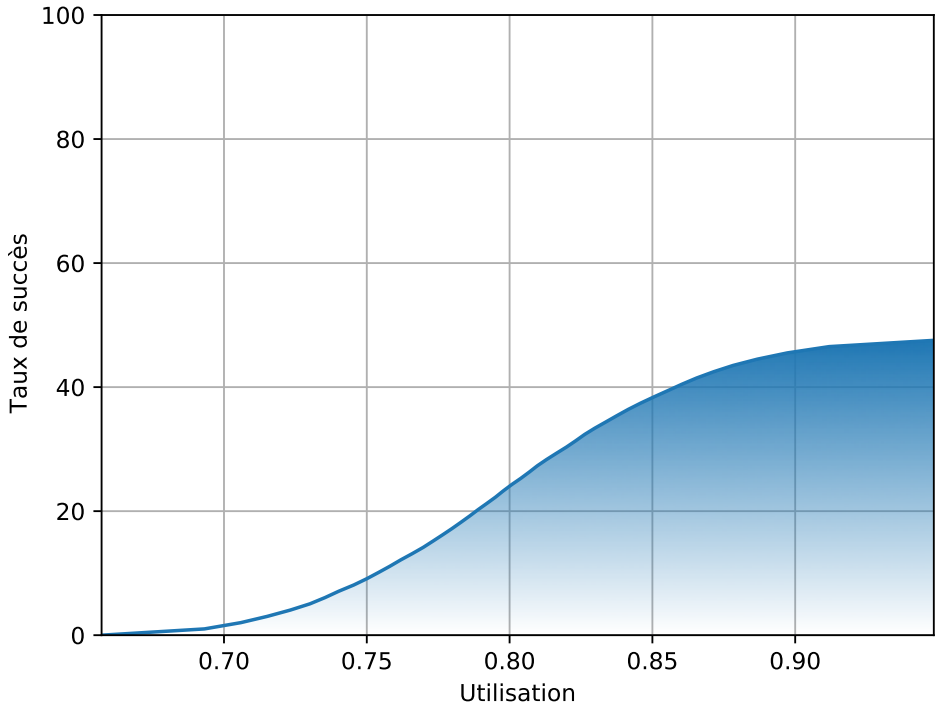


(A) Taux de succès de la combinaison des algorithmes proportionnels itératifs et harmoniques en fonction de l'utilisation avec FP au niveau des tâches

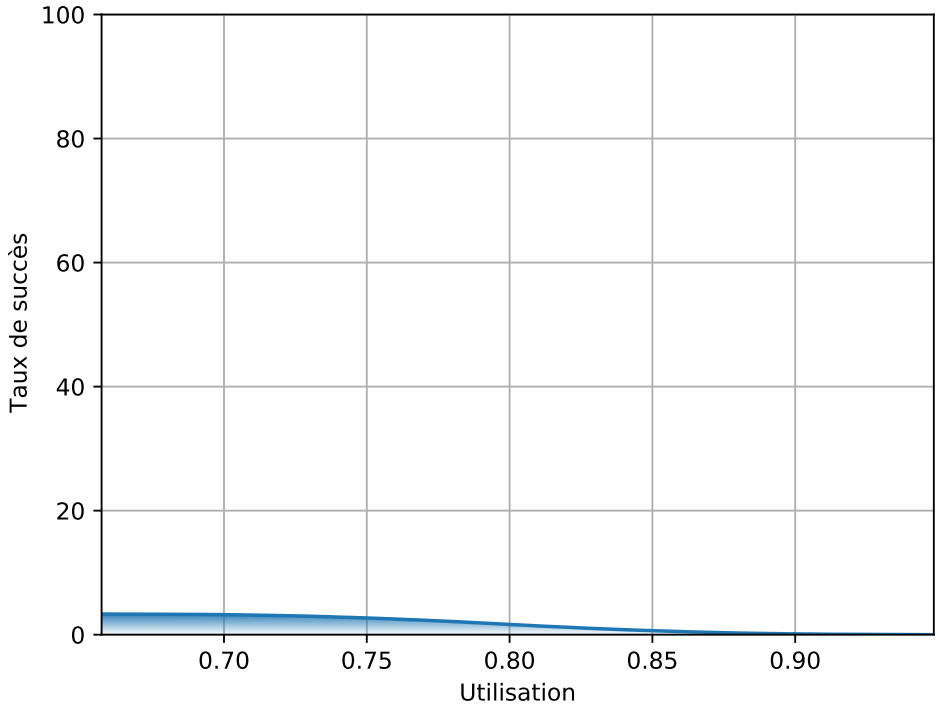


(B) Taux de succès de la combinaison des algorithmes proportionnels itératifs et harmoniques en fonction de l'utilisation avec DP au niveau des tâches

FIGURE 6.15 : Comparaison entre de la priorité fixe et de Dual Priority pour le second niveau et la combinaison de l'algorithme proportionnel itératif et harmonique pour le premier niveau.



(A) Taux de succès d'exactement deux VMs en fonction de l'utilisation avec DP au niveau des tâches sans overhead



(B) Taux de succès d'exactement deux VMs en fonction de l'utilisation avec DP au niveau des tâches avec overhead

FIGURE 6.16 : Comparaison entre le taux de succès d'exactement deux VMs sans et avec overhead.

Chapitre 7

Outils logiciels et expérimentations

Dans ce chapitre, nous présentons les différents développements logiciels effectués au cours de cette thèse. Parmi eux, nous avons eu l'occasion de développer un générateur de jeux de tâches servant de base à un simulateur d'ordonnancement temps réel. Le générateur nous permet ainsi d'obtenir des jeux de tâches aléatoires qui sont utilisés comme entrée pour le simulateur. Le simulateur nous a permis, quant à lui, de mesurer les performances en terme d'ordonnançabilité des différents algorithmes proposés dans ce manuscrit. Pour cela, nous avons mené des expérimentations qui nous ont permis d'obtenir le taux de succès de tous les algorithmes proposés. Nous avons ensuite intégré à ce simulateur un configurateur de machines virtuelles pour les besoins du projet FUI CEOS [136]. C'est d'ailleurs les contraintes architecturales imposées par ce projet qui ont conduit à ce que les simulations soient réalisées avec quatre VMs. Bien évidemment, les algorithmes proposés dans ce manuscrit permettent de trouver des solutions pour un nombre arbitraire de VMs. Dans le cadre du projet CEOS, les quatre VMs sont utilisées pour cloisonner quatre sous-sysèmes différents. La première embarque la partie contrôle-commande, la seconde embarque la partie en charge des communications, la troisième embarque la partie qui gère l'analyse du flux vidéo et la dernière embarque la partie navigation. Finalement, ce simulateur a permis de mettre en avant les avantanges et les inconvénients de ces différents algorithmes et plus globalement de leur intérêt dans un projet industriel.

Sommaire

7.1 Outil de génération de jeux de tâches

Dans cette section, nous présentons notre outil de génération de tâches aléatoires. Cet outil est utilisé comme entrée à l'outil de simulation d'ordonnancement présenté en détails dans la section suivante. Le but est ici de générer aléatoirement un très grand nombre de jeux de tâches et de les stocker dans une base de données. Cette base de données est ensuite classée pour que le simulateur puisse en extraire facilement des jeux de tâches selon certains critères tels que l'utilisation totale du processeur, en vue de tester nos algorithmes.

Pour la génération de tâches, nous avions utilisé dans un premier temps des algorithmes purement aléatoires. Mais le problème de ce type d'algorithmes est que même avec une période pour chaque tâche bornée à 1000, les hyper-périodes, notamment utilisées pour l'algorithme proportionnel itératif, ont tendance à très vite devenir exponentielles, ce qui donne des jeux de tâches inutilisables pour effectuer des simulations. Nous sommes donc ensuite revenus à l'algorithme le plus utilisé pour générer des jeux de tâches, à savoir UUnifast ([27]). Nous utilisons plus particulièrement une version adaptée dans [75] dans laquelle les périodes sont sélectionnées de manière à avoir une hyper-période bornée, tout en conservant le plus d'aléatoire possible. Pour ne pas biaiser l'aléatoire, tous les jeux de tâches avec des périodes multiples deux à deux entre elles (donc harmoniques) ont été supprimés. Cet ajout a été fait pour combler la version de UUnifast utilisée qui a tendance à créer plus de jeux de tâches harmoniques qu'on ne devrait normalement en trouver. En effet, ceux-ci étant censés être rares dans les applications réelles, et nos algorithmes fonctionnant particulièrement bien dans ce cas précis, ils ne seront pas pris en compte, pour avoir des résultats les plus objectifs possibles.

L'algorithme UUnifast est défini dans l'Algorithme 18. Cet algorithme prend en entrée le nombre n de tâches à créer et l'utilisation souhaitée pour le jeu de tâches. Son rôle est de répartir l'utilisation accordée à chaque tâche de manière aléatoire.

Algorithme 18 : UUnifast

Entrées : n, le nombre de tâches souhaitées dans le jeu de tâches, U, l'utilisation totale souhaitée pour les tâches
Sorties : $utilizationArray$, un tableau contenant les utilisations de chaque tâche

```
1 sumU = U;
2 utilizationArray[n] = 0;
3 pour i = 1; i < n − 1; i = i + 1; faire
4     nextSumU = sumU × aleatoire(0, 1)^{1/(n−i)};
5     utilizationArray[i] = sumU − nextSumU;
6     sumU = nextSumU;
7 retourner utilizationArray
```

À partir de cet algorithme, [75] propose de déduire les paramètres des tâches en fonction de leur utilisation. Pour cela, nous utilisons des matrices d'entiers premiers, puis nous multiplions des éléments aléatoires de cette matrice entre eux pour obtenir une période. Le WCET de chaque tâche est ensuite déduit de ces périodes et des utilisations (générées par UUnifast). Pour les besoins de nos simulations, les périodes sont ici bornées entre 100 et 1000, et les hyper-périodes sont, la plupart du temps, drastiquement réduites grâce à la façon de générer les périodes. En effet,

Tâche	C_i	T_i	D_i
τ_1	5	231	231
τ_2	23	462	462
τ_3	13	525	525
τ_4	8	700	700

TABLE 7.1 : Exemple de jeu de tâches généré aléatoirement dans notre base de donnée MongoDB

puisque les mêmes entiers premiers sont régulièrement utilisés pour définir les périodes, les chances d'obtenir des périodes avec un plus petit commun multiple faible sont grandement augmentées.

Nous nous sommes focalisés sur la génération de jeux de tâches ayant une utilisation totale entre 0.1 et 0.2, 0.2 et 0.3, 0.3 et 0.4 et 0.4 et 0.5. En effet, puisque nous utilisons quatre VMs lors des simulations, de plus grandes utilisations auraient été inutiles. Le nombre de tâches générées varie entre 2 et 6 par VM. Ce nombre est resté borné et faible pour, encore une fois, ne pas obtenir d'hyper-périodes trop élevées. Sur un seul jeu de tâches les hyper-périodes sont habituellement faibles, mais il faut prendre en compte ici que l'hyper-période se calcule sur le système entier, à savoir sur les quatre jeux de tâches associés aux VMs. Les jeux de tâches avec une hyper-période supérieure à 30000 ne sont pas sauvegardés, celles-ci étant déjà trop grandes pour effectuer des simulations dans un temps raisonnable pour une seule VM. De plus, nous avons réussi à obtenir un grand nombre de jeux de tâches différents avec une hyper-période bornée (plus de 15000000).

Tous les jeux de tâches générés sont ainsi conservés dans une base de données dédiée, utilisant MongoDB[1]. Cela permet de conserver une certaine souplesse sur les données conservées, la base de données étant NoSQL[2]. L'utilité principale de cette base est de pouvoir, lors des simulations, sélectionner quatre jeux de tâches aléatoirement répondant aux conditions voulues et évaluer l'efficacité des algorithmes de manière la moins biaisée possible. Les utilisations retenues pour la plupart des simulations sont comprises entre 0.1 et 0.2 d'utilisation pour les jeux de tâches des trois premières VMs et une utilisation allant de 0.3 à 0.4 pour la dernière. Nous présentons dans la Table 7.1 un exemple de jeu de tâches généré par la méthode présentée, et enregistré dans la base données MongoDB.

Ce sont tous les jeux de tâches ainsi générés qui nous permettront de n'avoir qu'à tirer aléatoirement des jeux de tâches dans cette base de données pour procéder aux simulations.

7.2 Outil de simulation d'ordonnancement temps réel

Avant de présenter notre simulateur, nous commençons par un bref historique de son développement, de son objectif et de ses différentes fonctions. Nous détaillons ensuite l'algorithme utilisé pour les simulations présentées dans ce manuscrit.

[1] https://www.mongodb.com/fr

[2] Not Only SQL est une classe de base de données opposée aux bases de données relationnelles

7.2.1 Historique de l'outil de simulation

Une première version du simulateur a été développée avant cette thèse. En effet, un premier stage en Master 1 consistait à développer un simulateur d'ordonnancement temps réel qui se voulait le plus générique possible. L'objectif était ainsi de pouvoir ajouter facilement différents algorithmes et différentes approches liées aux algorithmes temps réel. Pour parvenir à un tel simulateur, le premier objectif a été d'étudier ce qui pouvait se faire dans le domaine de l'ordonnancement temps réel. Le projet consistait donc, après lecture de différents états de l'art, à d'abord déterminer un design logiciel le plus générique possible.

Le simulateur d'ordonnancement temps réel a donc été développé en Java, langage choisi pour sa portabilité. De plus, ce langage est largement utilisé et comme il est orienté objet, cela facilite le développement d'un outil générique. Les premières fonctionnalités étaient ainsi diverses : le simulateur devait supporter différents types d'algorithmes d'ordonnancement, supporter les architectures mono-cœur et multi-cœurs, et supporter le plus d'approches possibles. Il était ainsi possible d'ajouter des algorithmes en-ligne, hors-ligne, de changer l'ordonnanceur, de créer des algorithmes, de créer des jeux de tâches, de créer de nouveaux modèles de tâches (en plus des modèles périodiques et sporadiques déjà implémentés). Ce simulateur devait permettre également de prendre en compte des paramètres liés à l'architecture, tels que les consommations énergétiques des tâches, ainsi que la recharge possible des batteries, pour supporter les algorithmes prenant en compte la consommation énergétique pour l'optimiser.

Une fois cette base réalisée, la généricité et la simplicité du simulateur ont été testées par un ensemble d'étudiants de Master 2. En effet, le simulateur a été proposé comme base pour des projets lors de ma dernière année d'étude de Master. Le but était alors de le mettre à l'épreuve, en permettant à tout un groupe de personnes de l'utiliser et de le compléter en ajoutant de nouvelles fonctionnalités et de nouvelles approches. Il s'agissait alors de revoir le design logiciel si nécessaire, à l'aide de regards extérieurs pour déterminer si tout était fonctionnel. Lors de ces essais, différentes fonctionnalités pour prendre en compte l'architecture matérielle ont été ajoutés par différentes personnes, comme par exemple les caches classiques, les caches à mémoire partagée, les caches à mémoire semi-partagée, les coûts de migration des tâches d'un cœur à l'autre dans un contexte multi-processeurs.

Afin de ne pas avoir à changer le code du simulateur pour changer les simulations, nous avons ajouté des parsers Json[3] pour prendre en compte tous ces ajouts, choisir le jeu de tâches, l'architecture matérielle et l'algorithme utilisé. À ce moment-là, aucun nouvel algorithme n'a été ajouté, le but était uniquement de reprendre une multitude d'algorithmes déjà existants, utilisant des approches les plus diversifiées possibles, et de les ajouter au simulateur. De plus, le code du simulateur était disponible sur un dépôt GitHub pour que le travail puisse être partagé. Mais ce travail n'étant qu'un projet étudiant, le dépôt est resté privé. Le simulateur aurait nécessité une plus grande documentation et un maintien régulier pour pouvoir être mis en ligne publiquement.

Ce simulateur a ensuite été modifié pendant la thèse pour supporter le modèle hiérarchique à deux niveaux. Parmi ces modifications, nous avons supprimé tous les éléments architecturaux et les modèles de tâches non utilisés.

Ensuite, nous y avons intégré progressivement les différents algorithmes proposés au cours de cette thèse pour vérifier leur fonctionnement, mais également pour pouvoir déterminer leur efficacité et finalement les comparer. Pour cela une base de

[3] https://www.json.org/json-fr.html

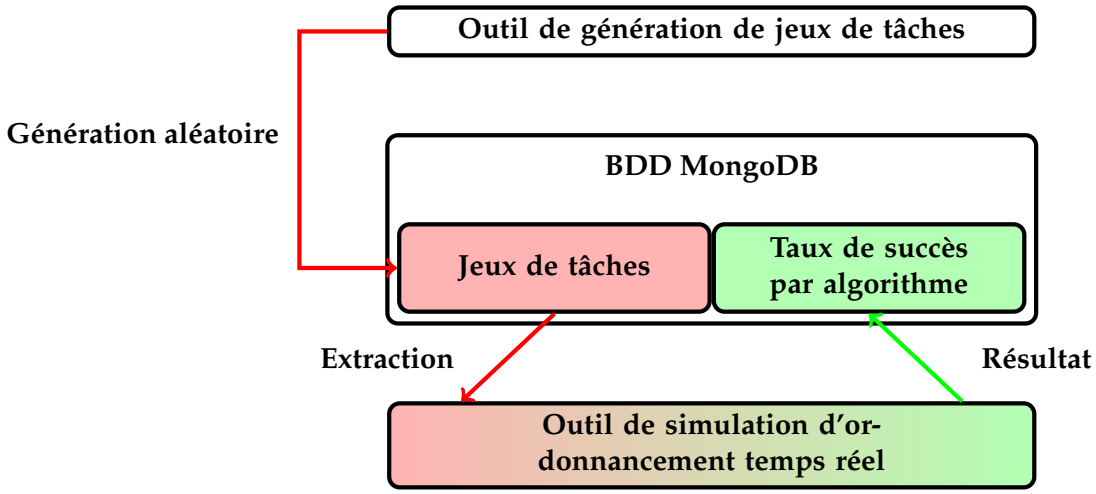


FIGURE 7.1 : Liens entre le générateur de tâches, le simulateur et la base de données MongoDB

données a été créée pour stocker tous les résultats. Comme expliqué dans la Section 7.1, le choix s'est porté sur le framework opensource MongoDB pour sa souplesse d'utilisation, son fonctionnement sans modèle prédéfini qui permet d'ajouter facilement des éléments à ce qui est déjà enregistré dans la base de données. Nous l'avons également choisi pour son interface avec Java, qui permet de convertir des classes Java entières en objet de la base de données. Toutes les manipulations de la base de données peuvent ainsi se faire directement en Java depuis le simulateur. Nous montrons dans la Figure 7.1 comment sont agencées les différentes parties des outils développés au cours de cette thèse.

Après avoir ajouté ce modèle et ces différentes fonctionnalités, la dernière étape du simulateur a été d'ajouter l'algorithme d'ordonnancement Dual Priority. Grâce aux choix de généricité précédent, peu de changements au sein du simulateur ont dû être effectués.

Comme nous l'avons vu dans les chapitres précédents, notamment pour la Dual Priority (Chapitre 6), certains des algorithmes utilisés peuvent avoir besoin de beaucoup de ressources que ce soit en puissance de calcul ou en mémoire vive. De plus, pour pouvoir valider nos algorithmes, il faut les appliquer sur de nombreux exemples, et donc de nombreux jeux de tâches. Cela entraîne donc des temps de simulation excessifs sur des machines équipées d'un processeur standard et d'une relativement faible quantité de mémoire. Nous avons dû utiliser des machines plus puissantes, dédiées aux calculs hautes performances, pour pouvoir lancer plus de simulations et obtenir des résultats bien plus rapidement. Certains calculs se sont en effet avérés impossibles avec une mémoire vive inférieure à 16 Go et d'autres pouvaient prendre plusieurs jours, voire des semaines. Nous sommes ainsi passés de machines de bureau avec en général 8 cœurs et 16 giga octets de Random Access Memory (RAM) à des serveurs allant jusqu'à 64 cœurs et 256 giga octets de RAM.

Mais un tel passage à l'échelle a soulevé de nouveaux problèmes. Ainsi, la plupart des classes ont dues être modifiées pour ne plus dépasser les limites mémoires de la machine virtuelle Java. Par exemple, certaines d'entre-elles enregistraient l'ordonnancement complet de toutes les tâches dans une variable pour l'afficher par la suite. En effet, nous rappelons que le premier objectif du simulateur était de valider les algorithmes d'ordonnancement et donc de conserver les ordonnancements produits (que ce soit pour les VMs ou les tâches ou sein des VMs). Ici, la conservation

de tous ces éléments nécessitaient trop de mémoire RAM pour conserver jusqu'à l'hyper-période, surtout en allant jusqu'à 64 simulations simultanées. De plus, le but n'était plus de trouver ces ordonnancements mais uniquement d'indiquer si l'ordonnancement était faisable ou non. Le simulateur a donc été modifié pour ne plus conserver toutes ces données qui devenaient considérables sur de grandes hyper-périodes.

Il a ainsi été nécessaire de revoir toutes les empreintes mémoire des différentes classes utilisées pour les réduire au minimum. Ces améliorations auraient pu être poussées à l'extrême, mais cela aurait été trop coûteux en temps de développement, en plus de rompre le contrat de base du simulateur, qui était d'être lisible et générique. En effet, pour réduire encore plus les empreintes mémoires, nous aurions pu ne plus passer par les différentes classes déjà construites, et tout refaire pour pouvoir libérer la mémoire dès que possible, et recalculer toutes les données au fur et à mesure, uniquement lorsque celles-ci sont nécessaires. L'inconvénient d'une telle façon de simuler est la lisibilité du code, mais également l'utilisation des différents cœurs des Central Processing Unit (CPU) qui aurait été décuplée à chaque nouveau calcul des données lorsqu'elles sont à nouveau nécessaires.

La version finale du simulateur contient 26630 lignes de code (sans compter les lignes vides ni les commentaires) réparties en 200 classes différentes. Actuellement, le simulateur est réservé à un usage interne, mais il serait possible d'envisager de le rendre public. Pour cela, il faudrait d'abord procéder à un nouveau design pour certaines parties, qui n'étaient destinées qu'à être utilisées qu'une seule fois, comme par exemple certaines simulations. Un code pour de simples simulations peut représenter des classes spécifiques pour accéder à la base de données, en extraire les données, en insérer de nouvelles dans une base de données résultats, et bien d'autres. Tous ces détails, avec en plus un manque de temps pour effectuer des patchs réguliers selon les retours des utilisateurs nous ont poussé à le restreindre à un usage interne. Un dernier développement a été nécessaire pour lancer automatiquement les simulations et obtenir les différents résultats de cette thèse.

7.2.2 Algorithme de simulation

Pour cela, nous avons développé l'algorithme présenté ci-dessous dans l'Algorithme 19, présenté en trois parties. Cet algorithme utilise une structure supplémentaire : ρ pour définir les VMs. Cette structure contient les attributs suivants :

- ρ représente une liste de quatre VMs.
- ρ^i représente la $i^{éme}$ VM de la liste.
- $\rho^i.taskset$ représente le jeu de tâches associé à la $i^{éme}$ VM.
- $\rho^i.ordo$ est une variable booléenne représentant si le jeu de tâches de la $i^{éme}$ VM est ordonnançable. Ce booléen est initialisé à faux.
- $\rho^i.U$ représente l'utilisation donnée à la VM qui sera utilisée par les différents algorithmes. Si les algorithmes n'ont pas besoin de cette valeur, elle n'est pas forcément définie. Cette variable est utilisée principalement par l'algorithme proportionnel itératif.

Nous résumons le fonctionnement de l'algorithme dans la Figure 7.2. Dans cette figure, nous montrons à nouveau la base de données MongoDB, qui nous sert d'entrée pour les simulations. En effet, la première étape est d'extraire quatre jeux de

tâches aléatoires de cette base de données, pour ensuite effectuer la première simulation. La première simulation est la plus complète : elle permet de tester la combinaison de tous les algorithmes précédemment définis. Dans cette première simulation, nous testons donc l'approche harmonique, qui nous donne un ordonnancement pour le plus de VMs possibles. Nous testons ensuite ces ordonnancements, avec la priorité fixe (FP dans la figure) et la Dual Priority (DP dans la figure). Pour toutes les VMs qui n'ont pas pu avoir un ordonnancement respectant les échéances des tâches, nous appliquons ensuite l'algorithme proportionnel itératif (API dans la figure), ce qui nous donne à nouveau des ordonnancements pour les VMs, que nous testons en priorité fixe et en Dual Priority au niveau des tâches. Nous obtenons ainsi toutes les VMs avec les ordonnancements associés si les tâches respectent leurs échéances pour chaque algorithme. Nous enregistrons alors ces informations dans la base de données MongoDB. Pour la seconde simulation, il s'agit des mêmes étapes, mais sans l'approche harmonique.

Simulation 1 : Cet algorithme commence par extraire aléatoirement quatre jeux de tâches de la base de données présentée dans la Section 7.1 (Ligne 1). ρ contient les quatre jeux de tâches qui seront attribués à chaque VM. En effet, pour les besoins des simulations et que celles-ci restent représentatives d'un système réel, nous n'avons sélectionné aléatoirement que quatre jeux de tâches, pour quatre VMs. L'utilisation de chaque jeu de tâches est également fixée par la base de données.

La Ligne 2 initialise un tableau contenant les overheads pour les VMs. Nous choisissons ici seulement les valeurs 0 (absence d'overhead) et 1. Nous effectuons ici une simplification par rapport à tous les cas possibles : il serait intéressant de fixer des overheads différents sur chaque VM, mais pour simplifier les simulations et surtout économiser en temps de calcul, lors de ces simulations, nous avons plutôt opté pour ces deux cas seulement.

La Ligne 3 exécute donc une simulation complète pour chaque overhead souhaité. À la Ligne 4 nous fixons un facteur α qui définit le pourcentage maximal que peut représenter l'overhead par rapport à la période d'une VM. Par exemple, avec un overhead donné de 1, la période minimale de la VM est alors fixée à 50, d'après l'Équation (3.23) (Page 72).

Ayant fixé les paramètres des VMs, nous pouvons maintenant essayer d'ordonnancer le plus de VMs possibles de la liste de manière harmonique. Pour cela, nous devons commencer par déterminer les périodes minimales (Ligne 5) et maximales (Ligne 6) de la première VM. À l'aide de ces bornes, nous pouvons ensuite calculer les paramètres de la première VM (Ligne 7). C'est-à-dire son temps d'exécution C^1, sa période T^1, ainsi qu'une variable booléenne *sched* indiquant si les tâches de la VM respectent leurs échéances ou non.

Nous initialisons ensuite une liste vide (Ligne 8) qui nous permettra de sauvegarder toutes les VMs qui ont pu être ordonnancées par l'algorithme harmonique et dont toutes les tâches respectent leurs échéances. Avant de vérifier cela, nous devons d'abord vérifier que la première VM a pu être ordonnancée, et que toutes ses tâches respectent leurs échéances (Ligne 9). Si tel est le cas, nous pouvons alors ajouter la première VM dans la liste des VMs ordonnançables en mode harmonique (Ligne 10), puis chercher les paramètres (durée d'exécution et période) des autres VMs de manière harmonique (Ligne 11). Nous créons ensuite trois listes destinées à contenir des VMs (Ligne 25). Ces listes nous serviront à déterminer quelles VMs ont pu être ordonnancées de manière harmonique et dont toutes les tâches respectent leurs échéances en utilisant RM, mais également celles qui n'ont pas pu l'être et, enfin, celles dont les tâches respectent leurs échéances grâce à la Dual Priority.

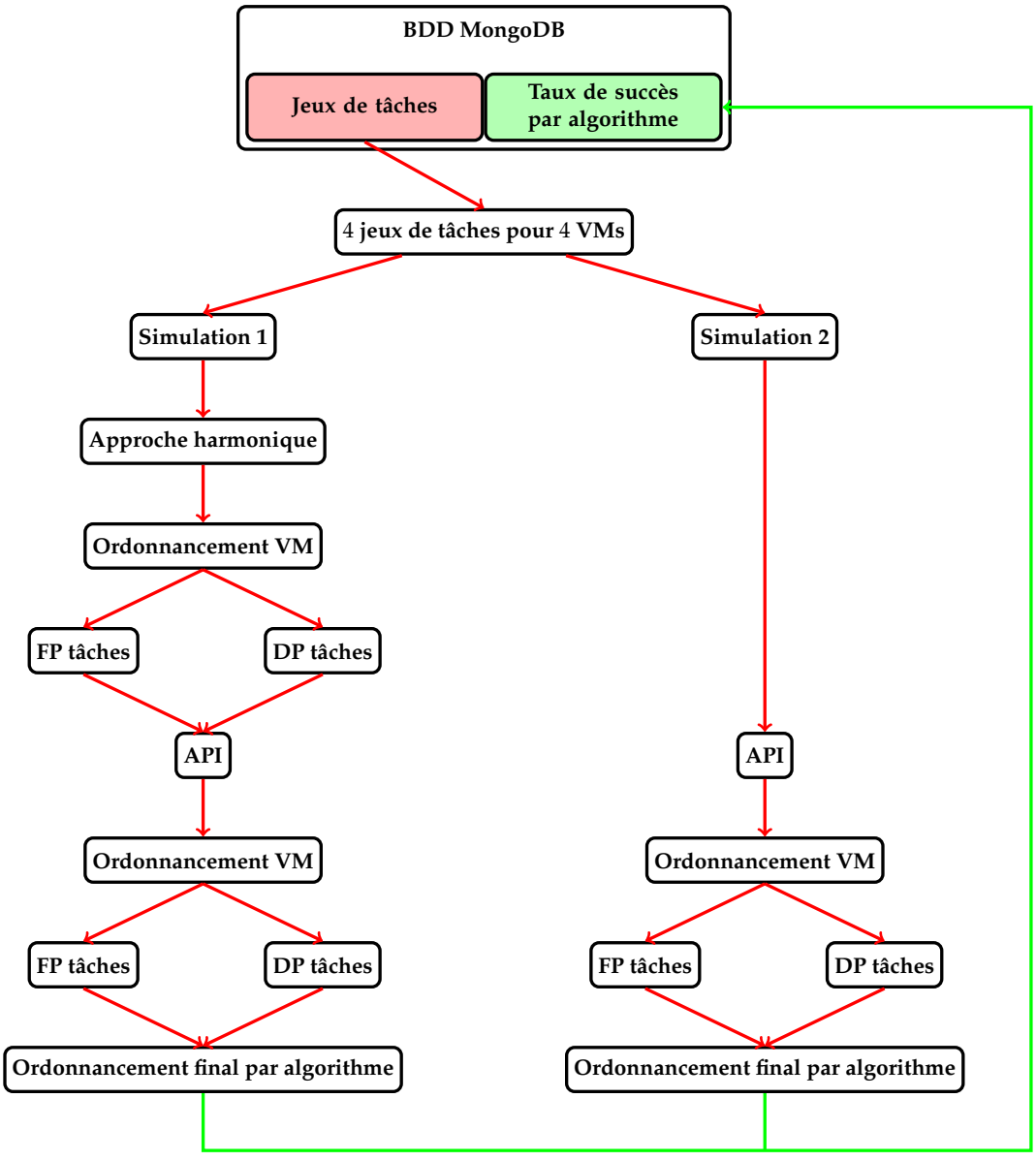


FIGURE 7.2 : Fonctionnement de l'algorithme de simulation

Algorithme 19 : Algorithme final utilisé pour les simulations (1/3)

```
1  ρ = extractionAléatoire(Base de données MongoDB);
2  entier[] overheads = {0, 1}; // Il est possible de simuler pour plus de
     valeurs d'overheads
3  pour chaque overhead ∈ overheads faire
4      α = 0.02;
5      T^1_min = Période minimale de la première VM donnée par l'Équation (3.25);
         // Équation Page 73
6      T^1_max = Période maximale de la première VM donnée par
         l'Équation (3.31); // Équation Page 74
7      ρ^1(C^1), ρ^1(T^1), sched = Algorithme 2(T^1_min, T^1_max, ρ^1.taskset);
         // Paramètres de la première VM (algorithme Page 78)
8      listeVMHarmo = ∅;
9      si sched alors
10         listeVM.add(ρ^1);
11         listeVM = Algorithme 3(ρ); // Paramètres des VM harmoniques
             (algorithme Page 79)
12         listeVMHarmoFP, listeVMNonHarmoFP, listeVMHarmoDP = ∅;
13         ordoVMs = Algorithme 4(ρ^1, listeVM); // Ordonnancement des VMs
             harmoniques (algorithme Page 83)
14         pour chaque ρ^i ∈ listeVM faire
15             si ρ^i ordonnançable d'après Équation (3.12) appliquée sur ordoVMs[i];
                 // Équation Page 70
16             alors
17                 listeVMHarmoFP.add(ρ^i);
18             sinon
19                 listeVMNonHarmoFP.add(ρ^i);
20         pour chaque ρ^i ∈ listeVMNonHarmoFP faire
21             τ = Algorithme 15(ρ^i.taskset, ρ^i.ordo); // Test d'ordonnancement
                 Dual Priority (algorithme Page 137)
22             si τ ≠ ∅ alors
23                 ρ^i.taskset = τ;
24                 listeVMHarmoDP.add(ρ^i);
25     listeVMHarmo.add(listeVMHarmoFP ⋃ listeVMHarmoDP);
26     listeVMNonHarmo = ρ \ listeVMHarmo;
```

Algorithme final utilisé pour les simulations (2/3)

```
27  pour chaque ρ^i ∈ listeVMNonHarmo faire
28      ρ^i.U = Calcul de l'utilisation accordée à la VM via l'Équation (5.14);
          // Équation Page 108
29  tmax = ppcm(listeVMNonHarmo.U, listeVMHarmo);
30  booléen déjàOrdo[0 ... tmax] = conversion(∀ρ^i.ordo ∈ listeVMHarmo,
      tmax);
31  listeVMPfairFP, listeVMPfairDP, listeVMNonPfairFP = ∅;
32  pour chaque ρ^i ∈ listeVMNonHarmo faire
33      booléen ordoVM[0 ... tmax] = Algorithme 9(ρ^i.U, déjàOrdo, tmax);
          // Algorithme proportionnel itératif (algorithme Page 107)
34      ordonnançable = simulationFP(ρ^i, ordoVM);
35      si ordonnançable alors
36          listeVMPfairFP.add(ρ^i);
37          déjàOrdo.add(ordoVM);
38      sinon
39          τ = Algorithme 15(ρ^i.taskset, ρ^i.ordo); // Test d'ordonnancement
              Dual Priority (algorithme Page 137)
40          si τ ≠ ∅ alors
41              ρ^i.taskset = τ;
42              listeVMPfairDP.add(ρ^i);
43              déjàOrdo.add(ordoVM);
44
45
46
```

Algorithme final utilisé pour les simulations (3/3)

```
47  pour chaque ρ^i ∈ ρ faire
48      ρ^i.U = Calcul de l'utilisation accordée à la VM via l'Équation (5.14);
          // Équation Page 108
49  tmax = ppcm(ρ);
50  listeVMPfairSeulFP = ∅;
51  listeVMPfairSeulDP = ∅;
52  booléen déjàOrdo[];
53  pour chaque ρ^i ∈ ρ faire
54      booléen ordoVM[] = Algorithme 9(ρ^i.U, déjàOrdo, tmax);
          // Algorithme proportionnel itératif (algorithme Page 107)
55      ordonnançable = simulationFP(ρ^i, ordoVM);
56      si ordonnançable alors
57          listeVMPfairSeulFP.add(ρ^i);
58          déjàOrdo.add(ordoVM);
59      sinon
60          τ = Algorithme 15(ρ^i.taskset, ρ^i.ordo); // Test d'ordonnancement
              Dual Priority (algorithme Page 137)
61          si τ ≠ ∅ alors
62              ρ^i.taskset = τ;
63              listeVMPfairSeulDP.add(ρ^i);
64              déjàOrdo.add(ordoVM);
65
66
67
```

Nous cherchons ensuite l'ordonnancement des VMs de façon à ce qu'elles soient ordonnancées de manière harmonique (Ligne 13). La liste $ordoVMs$ contient alors pour chaque VM son ordonnancement complet. Une fois l'ordonnancement trouvé, nous testons pour chaque VM ordonnancée de manière harmonique (Ligne 14) si toutes les tâches de cette VM respectent leurs échéances en utilisant un algorithme de priorité fixe, puis le temps de réponse donné par l'Équation (3.12) (Page 70) appliqué à l'ordonnancement de la VM étudiée (Ligne 16). Si cet ordonnancement est trouvé, la VM est alors ajoutée à la liste des VMs ordonnançables de manière harmonique ($listeVMHarmoFP$) en utilisant la priorité fixe (Ligne 17). Sinon, la VM est ajoutée à la liste des VMs dont les tâches ne sont pas ordonnançables ($listeVMNonHarmoFP$) en utilisant un algorithme de priorité fixe (Ligne 19).

La suite consiste à tester, pour chaque VM dont un ordonnancement harmonique a été trouvé mais dont les tâches ne respectent pas leurs échéances en priorité fixe (Ligne 20), si les tâches peuvent être ordonnancées utilisant la Dual Priority (Ligne 21) et donc avoir de nouveaux paramètres associés. Si tel est le cas (Ligne 22), le jeu de tâches, avec de nouveaux paramètres répondant à la Dual Priority, est actualisé (Ligne 23) et la VM est ajoutée à la liste des VMs harmoniques dont les tâches sont ordonnancées à l'aide de la Dual Priority (Ligne 24).

À ce stade, nous avons connaissance de toutes les VMs qui ont pu être ordonnancées de manière harmonique et dont les tâches respectent leurs échéances, que ce soit grâce à de la priorité fixe ou la Dual Priority. Nous ajoutons ainsi toutes ces VMs à la liste des VMs ordonnancées de manière harmonique (Ligne 25). Nous actualisons également la liste complémentaire à celle-ci, contenant toutes les VMs qui n'ont soit pas pu être ordonnancées de manière harmonique, soit dont les tâches n'ont pas pu respecter leurs échéances (Ligne 26).

Passons à la seconde partie de l'algorithme. Nous pouvons maintenant appliquer l'algorithme proportionnel itératif sur toutes les VMs qui n'ont pas pu être ordonnancées de manière harmonique. Pour commencer, nous devons d'abord parcourir toutes les VMs dans la liste des VMs non harmoniques (Ligne 27, comme défini dans le Chapitre 5, Équation (5.14), Page 108) et déterminer pour chacune leur utilisation attribuée (Ligne 28).

Ensuite, il est possible de démarrer l'algorithme proportionnel itératif. Pour cela, trois variables sont d'abord initialisées.

La première variable initialisée, $tmax$ (Ligne 29), est calculée à partir du plus petit commun multiplicateur des périodes des VMs déjà ordonnancées et du dénominateur de l'utilisation des VMs à ordonnancer. Cela nous permettra de déduire le temps $tmax$ sur lequel les VMs doivent être ordonnancées pour des résultats optimaux.

La seconde variable à initialiser est un tableau de booléen appelé $déjàOrdo$ (Ligne 30). Ce tableau indique pour chaque pas de temps, jusqu'à l'instant $tmax$, si une VM est déjà ordonnancée (valeur à $VRAI$) ou non (valeur à $FAUX$). Les ordonnancements étant précédemment donnés dans des intervalles de temps, pour des raisons de simplicité et de lisibilité, une conversion est alors nécessaire. Cette conversion prend en entrée tous les ordonnancements de toutes les VMs déjà ordonnancées et indique à chaque instant si une VM est ordonnancée. Dans la suite, ce tableau sera actualisé à chaque nouvelle VM ordonnancée par l'algorithme.

Trois listes vides sont ensuite initialisées (Ligne 31) : $listeVMPfairFP$, $listeVMPfairDP$ et $listeVMNonPfairFP$. Elles contiennent respectivement la liste des VMs qui ont pu être ordonnancées par l'algorithme proportionnel itératif et dont les tâches respectent leurs échéances en étant ordonnancées par RM, les VMs ordonnancées dont les tâches respectent leurs échéances grâce à RM, et la liste des VMs dont les tâches ne respectent pas leurs échéances via RM.

La suite, Ligne 32 à Ligne 37 est l'algorithme proportionnel itératif lui-même. Il est ainsi appliqué à chaque VM pour laquelle un ordonnancement harmonique n'a pas été trouvé ou pour laquelle les tâches n'ont pas pu respecter leurs échéances (Ligne 32). Un tableau booléen contenant l'ordonnancement de chaque VM est ensuite calculé par l'algorithme proportionnel itératif (Ligne 33). Une simulation est ensuite effectuée (comme définie dans le Chapitre 6) pour vérifier si les tâches de cette VM, avec l'ordonnancement trouvé, respectent leurs échéances en utilisant RM comme algorithme d'ordonnancement pour les tâches (Ligne 34). Si toutes les tâches de la VM étudiée sont ordonnançables (Ligne 35), la VM est alors ajoutée à la liste des VMs pour lesquelles un ordonnancement proportionnel itératif est trouvé et dans laquelle toutes les tâches respectent leurs échéances grâce à RM (Ligne 36). La variable *déjàOrdo* est ensuite actualisée (Ligne 37), puisqu'une nouvelle VM est traitée.

En revanche, si toutes les tâches n'étaient pas ordonnançables par RM, il faut alors appliquer l'algorithme Dual Priority (Ligne 39). Si toutes les tâches sont ordonnançables en utilisant cet algorithme d'ordonnancement, le jeu de tâches de la VM est alors mis à jour avec les nouveaux paramètres (Ligne 41), la VM est ajoutée à la liste des VMs ordonnançables avec l'algorithme proportionnel itératif et Dual Priority pour le niveau des tâches (Ligne 42) et la variable *déjàOrdo* est actualisée (Ligne 43).

Simulation 2 : La suite (Lignes 47 à 64) de l'algorithme consiste à effectuer à nouveau les étapes à partir de la Ligne 27, mais cette fois en ne considérant plus les VMs déjà ordonnancées par l'algorithme harmoniques, nous testons uniquement les résultats de l'algorithme proportionnel itératif seul. Puisqu'aucune différence particulière n'est à noter, nous n'entrerons pas plus dans les détails sur cette partie de l'algorithme.

Grâce à cet algorithme, nous avons pu mener à bien des expériences pour obtenir différents taux de succès, grâce aux différentes listes utilisées dans l'algorithme que nous avons ensuite enregistrées dans une base de données, pour en extraire les différents taux de succès.

7.2.3 Expériences

Comme indiqué dans la Section 4.4 du Chapitre 4, dans la Section 5.4 du Chapitre 5 et la Section 6.4 du Chapitre 6, c'est l'algorithme de la Section 7.2 qui est utilisé pour obtenir tous les résultats de simulations.

Parmi les résultats obtenus, nous avons pu obtenir des taux de succès en fonction de l'utilisation pour l'algorithme harmonique seul, avec la priorité fixe pour l'ordonnancement des tâches (Section 4.4 du Chapitre 4). Nous avons d'abord cherché à obtenir des résultats sur des systèmes chargés (avec une utilisation allant de 0.63 à 1) pour quatre VMs sans overhead. Nous avons ensuite cherché à comparer ces taux de succès en ajoutant les overheads sur les VMs. Mais avec un tel nombre de VMs et des systèmes autant chargés, l'augmentation de la charge impliquée par l'overhead réduit alors trop les taux de succès, et nous n'avons pu en observer que pour les cas où exactement une VM est ordonnancée, puis exactement deux VMs. Puisque le but de cet algorithme est de trouver un premier ordonnancement pour les VMs, les taux de succès restent très faibles. C'est pour cela que nous avons ensuite cherché à compléter l'ordonnancement trouvé par cet algorithme.

Nous avons ensuite déterminé les taux de succès de l'algorithme précédent complété par un second : l'algorithme proportionnel itératif (Section 5.4 du Chapitre 5). Cette fois-ci, nous avons pu obtenir un taux de succès s'appliquant sur toutes les

VMs et ainsi obtenir le taux de succès final de cette première solution. Pour des systèmes à quatre VMs aussi chargés, les premiers résultats sont satisfaisants : nous arrivons à un taux de succès de 65% pour une utilisation de 0.63 jusqu'à descendre à un taux de succès de 0% pour une utilisation de 0.9.

Nous avons donc cherché à aller plus loin, en intégrant la Dual Priority comme algorithme d'ordonnancement pour le second niveau, c'est-à-dire les tâches. Pour cela, nous avons utilisé l'algorithme FDMS. Nous avons ensuite obtenu de nouveaux taux de succès à l'aide de cette approche, en l'intégrant que ce soit sur la partie harmonique ou sur l'algorithme proportionnel itératif. Les taux de succès sont dans la Section 6.4 du Chapitre 6. C'est ici que nous obtenons les taux de succès les plus aboutis pour tous nos travaux. Nous obtenons ainsi un taux de succès de 90% pour une utilisation de 0.63 qui descend de manière linéaire jusqu'à un taux de succès de 0% proche d'une utilisation de 1. Pour un modèle hiérarchique à deux niveaux, cette combinaison est la meilleure obtenue au cours de cette thèse.

Avec de tels taux de succès et puisque cette thèse est liée au projet CEOS, nous avons cherché à produire un outil de configuration automatique des VMs à partir de nos algorithmes pour que nos algorithmes soient directement utilisés dans le projet. C'est ce que nous présentons dans la section suivante.

7.3 Outil de configuration des VMs

Dans cette section, nous présentons les différents travaux d'implémentation réalisés pour pouvoir utiliser les algorithmes développés précédemment dans une application réelle : celle du projet CEOS. Pour cela, nous commençons par définir les besoins du projet puis nous présentons les différentes fonctionnalités et options de l'outil développé.

Comme vu au début du Chapitre 1, le projet CEOS associe les industriels Thalès, Alerion, EDF, Enedis, Aéroport de Caen et les laboratoires de l'INRIA Nancy-Grand Est, Inria Paris et de l'ESIEE pour développer un drone autonome. Dans ce projet, le système embarqué par le drone repose sur trois catégories d'applications isolées par sécurité dans des VMs différentes. Ces VMs sont contrôlées par l'hyperviseur PikeOS. PikeOS est à la fois un système d'exploitation temps réel et un hyperviseur certifié avionique ARINC 653 et Multiple Independent Levels of Security (MILS) (voir Section 2.3.2 du Chapitre 2). Celui-ci permet d'utiliser un modèle hiérarchique à deux niveaux. En effet, PikeOS permet d'ordonnancer des VMs qui elles-mêmes, exécutent des tâches. La difficulté ici pour cet hyperviseur est qu'il faut déterminer les temps d'activation de chaque VM "à la main", sous la forme d'un schéma qui est répété dans le temps. Les tâches sont donc connues, mais il faut calculer l'ordonnancement des VMs. C'est ici que les résultats de cette thèse entrent en jeu, puisque nous pouvons calculer ces ordonnancements à l'aide de nos algorithmes implantés dans un outil spécifique. Cet outil, lui aussi, a été développé en Java, afin de pouvoir être utilisé le plus largement possible sur un grand nombre de plateformes.

Cet outil prend ainsi un fichier d'entrée, écrit en Json, pour plus de portabilité et de simplicité pour l'utilisateur, et produit l'ordonnancement des VMs. Le résultat est également produit sous la forme d'un fichier Json. Cela permet de le parser ensuite facilement pour configurer automatiquement CODEO, l'outil de configuration de PikeOS. Aucun autre paramètre supplémentaire n'est renseigné en dehors des fichiers Json d'entrée. Le but était de pouvoir utiliser ce programme facilement sans jamais avoir à modifier les sources.

L'outil contient donc un premier parser, qui permet de lire le Json donné en entrée. Ce Json contient les différents jeux de tâches des différentes VMs pour pouvoir ensuite en déduire un ordonnancement pour les VMs. Pour le projet CEOS une option a été ajoutée : la possibilité de ne fournir que les utilisations des VMs et non pas le jeu de tâches entier. En effet, dans les premiers mois, les tâches n'étaient pas connues.

Pour pallier les différents problèmes qui pourraient survenir en ne donnant que les utilisations des tâches, d'autres données sont requises. L'utilisateur doit spécifier la valeur de la période minimale et maximale pour chaque VM dont seule l'utilisation est donnée. En effet, l'algorithme déterminait automatiquement ces valeurs minimum et maximum en fonction du jeu de tâches associé. Cette étape supplémentaire est obligatoire pour tout jeu de tâches qui n'est pas spécifié mais dont seule l'utilisation est spécifiée.

D'autres options ont été ajoutées. Il est par exemple possible, lorsque le jeu de tâches est donné, de choisir si l'ordonnançabilité du jeu de tâches doit être testée. Une des fonctionnalités demandée était de pouvoir prendre en compte un nombre indéterminé de tâches, puisque leur nombre n'est pas toujours connu à l'avance.

L'outil étant demandé assez tôt dans la chronologie du projet CEOS, il y avait beaucoup d'inconnues, il fallait pouvoir anticiper le plus de comportements possibles tout en étant le plus souple possible. Dans ce but, notre outil permet de choisir les différents algorithmes utilisés. Ainsi, par défaut, l'outil ne va créer que des machines virtuelles avec des périodes harmoniques entre-elles (algorithmes du Chapitre 4). Mais comme nous l'avons vu, cet algorithme a un taux de succès faible, il a été pensé pour être complété par un autre algorithme, à savoir celui du Chapitre 5.

Il est également possible de définir les overheads des différentes VMs, s'ils sont connus. Ceux-ci pourront ensuite être pris en compte lors des vérifications d'ordonnancement des tâches, mais également directement dans les algorithmes eux-mêmes, comme par exemple, l'algorithme qui fonctionne avec les périodes harmoniques.

L'ajout des overheads implique également l'ajout d'un facteur α. Ce facteur α permet de définir le pourcentage maximum que peut prendre l'overhead d'une VM par rapport à sa période, comme nous l'avions vu dans le Chapitre 4.

Finalement, ce troisième outil permet d'appliquer et valider les algorithmes produits au cours de cette thèse sur un projet réel tout en les adaptant aux besoins du projet.

Chapitre 8

Conclusion et perspectives

Dans ce dernier chapitre, nous commençons par rappeler le modèle hiérarchique à deux niveaux utilisé au cours de cette thèse. Ce modèle a été choisi car il permet de représenter la manière dont les tâches sont mises en œuvre dans PikeOS. Nous rappelons ensuite les algorithmes présentés pour ordonnancer les VMs ainsi que les tâches en considérant ce modèle hiérarchique à deux niveaux. Enfin, nous envisageons les perspectives de recherche pour continuer ces travaux.

Dans ce manuscrit, nous avons apporté une solution à un problème général important : «comment trouver un ordonnancement pour des VMs afin de permettre d'ordonnancer le plus de tâches possibles au second niveau tout en respectant leurs échéances ?». Ce travail a été appliqué concrètement à l'hyperviseur PikeOS. Dans un premier temps, nous avons étudié le modèle hiérarchique à deux niveaux. Nous avons proposé une analyse d'ordonnançabilité pour un système de tâches ordonnancées avec ce modèle. Dans un second temps, nous nous sommes intéressés à proposer une approche d'ordonnancement pour les VMs qui soit la plus simple possible à mettre en œuvre. L'algorithme d'ordonnancement résultant de cette approche est basé sur un ordonnancement avec des VMs ayant des périodes harmoniques. Après avoir constaté que les performances en terme d'ordonnançabilité de cet algorithme ne permettaient pas d'obtenir un taux de succès suffisamment élevé (0% pour quatre VMs), nous avons proposé une seconde approche d'ordonnancement pour les VMs. Cette seconde approche complète la première dans le sens où elle trouve un ordonnancement pour les VMs qui n'ont pas pu être ordonnancées par la première approche. L'algorithme résultant de cette seconde approche d'ordonnancement est appelé algorithme proportionnel itératif. Le nombre de systèmes ordonnançables obtenu par cet algorithme est bien plus élevé que celui obtenu par le premier (environ 60% pour une utilisation de 0.63). Afin d'enrichir notre étude et utiliser les ressources matérielles au maximum, nous avons également considéré l'approche Dual Priority pour remplacer l'ordonnancement à priorités fixes pour les tâches ordonnancées dans les VMs. Afin de pouvoir comparer les performances en terme d'ordonnançabilité de ces différentes approches, nous avons développé un outil de simulation d'ordonnancement temps réel. Grâce à cet outil, nous avons pu simuler le fonctionnement de tous les algorithmes développés durant de cette thèse et mesurer les performances en terme d'ordonnançabilité de différentes combinaisons d'algorithmes. La conclusion de cette étude est que l'approche la plus efficace consiste en un ordonnancement harmonique pour les VMs, complété par l'algorithme proportionnel itératif et Dual Priority pour l'ordonnancement des tâches (nous atteignons presque 100% de taux de succès pour une utilisation de 0.63).

8.1 Modèle hiérarchique à deux niveaux

Pour la mise en œuvre de ce modèle, nous avons tout d'abord étudié le fonctionnement de l'hyperviseur PikeOS qui a été retenu de par le contexte applicatif du projet CEOS. Durant cette étude, nous nous sommes intéressés à des algorithmes fonctionnant avec un nombre variable de VMs. En effet, à ce stade de développement du projet CEOS, le nombre de VMs nécessaire n'avait pas encore été statué. Une fois les besoins applicatifs affinés, il s'est avéré qu'un système composé de quatre VMs était suffisant pour répondre au cahier des charges. C'est pourquoi nous avons réalisé nos tests en simulant des systèmes composés de quatre VMs.

PikeOS permet de spécifier une date de début et de fin d'exécution pour chacune des VM dont il a la charge. Néanmoins, aucun outil n'est fourni pour calculer le meilleur ordonnancement pour l'ensemble de ces VMs. Nous avons donc un nombre déterminé de VMs, ainsi que de tâches dans chaque VM. Les paramètres de ces tâches et de ces VMs sont connus et nous avons alors besoin d'un outil pour déterminer les dates de début et de fin d'exécution des VMs. L'ordonnancement des VMs n'étant paramétrable que par la spécification d'une table d'ordonnancement, nous avons besoin d'un algorithme hors-ligne nous permettant de la calculer.

L'originalité des travaux de cette thèse réside dans le fait que, jusqu'à maintenant, les modèles hiérarchiques étaient définis en considérant des serveurs de tâches et les études sur ces modèles se concentraient sur des approches d'ordonnancement en ligne. Dans ce manuscrit, nous considérons une approche d'ordonnancement hors-ligne et proposons différents algorithmes qui déterminent les dates d'exécution des VMs a priori.

Une première étape a été de définir formellement le modèle utilisé dans ce manuscrit. Il a ensuite fallu déterminer une condition d'ordonnançabilité pour une tâche s'exécutant au sein d'une VM, sachant que l'exécution de la tâche pouvait être interrompue lorsque l'exécution de la VM était interrompue par l'hyperviseur. Pour cela, nous avons déterminé les candidats d'instant critique des tâches qui permettent de connaître le pire cas d'activation pour chacune d'entre-elles. En les activant à ces instants précis, cela permet de s'assurer que si les tâches respectent leurs échéances pour ces scénarios, alors elles respectent leurs échéances dans tous les cas.

Une fois ces candidats d'instant critique définis, nous avons établi deux formules pour le calcul de temps de réponse : la première est une simplification de la seconde adaptée à des VMs ordonnancées de manière harmonique dont les tâches sont ordonnancées en priorité fixe. Grâce à l'harmonicité, nous nous assurons que la VM ne soit ordonnancée qu'une seule fois par période d'activation, et ne soit pas préemptée lors de son exécution.

La seconde formule de calcul de temps de réponse, quant à elle, fonctionne dans un motif arbitraire d'activation d'une VM. La seule contrainte est ici de connaître à l'avance l'ordonnancement de la VM étudiée. Elle peut en revanche être ordonnancée un nombre arbitraire de fois au cours de sa période. Cette formule est basée sur les candidats d'instant critique des tâches ordonnancées dans des VMs : il n'est pas possible de savoir quel candidat sera l'instant critique. Cette formule permet ainsi de tester les différents candidats afin de déterminer le temps de réponse pire cas des différentes tâches au sein de la VM. Une fois les pires temps de réponse calculés, nous pouvons déterminer si les tâches respectent leurs échéances.

À partir de ces différentes formules de temps de réponse, nous avons pu déterminer les conditions d'ordonnançabilité sur les VMs. Ces conditions nous permettent ainsi de déterminer les périodes minimales et maximales des VMs basées sur les tâches au sein des VMs. Nous avons également pris en compte les overheads sur les VMs. Ces overheads sont les coûts (en temps) de changement de contexte, c'est-à-dire lorsque le système passe de l'exécution d'une VM à une autre.

C'est grâce à toutes ces conditions d'ordonnançabilité que nous pouvons ensuite déterminer un premier ordonnancement pour les VMs prenant en compte les tâches associées à chaque VM.

8.2 Ordonnancement des VMs

Ce premier ordonnancement étudié n'est pas voué à ordonnancer toutes les VMs du système. Il permet en revanche d'en ordonnancer quelques-unes pour que les algorithmes suivants aient moins de VMs à ordonnancer, mais également pour obtenir des VMs avec les paramètres les plus simples possibles. De plus, les périodes des VMs étant harmoniques entre-elles, l'hyper-période découlant d'un tel système est bien plus restreinte qu'avec d'autres types d'ordonnancements. Cette propriété permet également d'obtenir des résultats plus rapides pour les algorithmes suivants.

L'algorithme d'ordonnancement proposé pour les premières VMs est un algorithme utilisant l'harmonicité des périodes. Nous avons ainsi dans un premier temps

défini ce qu'est l'harmonicité deux à deux entre les différentes périodes des VMs. Nous en avons déduit de nouvelles conditions d'ordonnançabilité spécifiques à des VMs harmoniques.

À partir de ces conditions, nous en avons déduit les paramètres de la première VM la plus critique, c'est-à-dire son temps d'exécution et sa période. Pour cela, nous avons utilisé une recherche dichotomique grâce aux bornes minimales et maximales sur la période et nous avons vérifié l'ordonnancement des tâches avec les paramètres trouvés. À partir des paramètres de la première VM, nous avons déduit les paramètres de toutes les autres VMs, par niveau de criticité décroissant, dans le cas d'une période d'activation harmonique des VMs. Nous avons ainsi cherché à obtenir le plus de VMs possibles avec des paramètres harmoniques.

À ce stade, nous n'avons déterminé que les paramètres des VMs harmoniques et non les ordonnancements à proprement parler. Nous avons également testé uniquement si les tâches de la première VM respectaient leurs échéances. Pour pouvoir ordonnancer les VMs, nous avons ainsi proposé un algorithme dérivé de l'état de l'art pour lequel aucune formalisation n'avait été précisée. Pour cela, nous avons mis en œuvre une approche couramment utilisée pour résoudre des problèmes d'optimisation : le «Bin Packing». Ce type d'algorithme est souvent utilisé en ordonnancement temps réel pour répondre à des problèmes d'ordonnancement multi-cœurs par partitionnement, mais nous l'avons adapté ici à un problème avec un seul cœur.

Comme dit précédemment, notre premier algorithme utilisant l'approche harmonique engendre un taux de succès qui n'est pas satisfaisant pour répondre aux contraintes du projet CEOS. En effet, pour quatre VMs il est de 0% et pour trois VMs il débute à 75% avec une utilisation totale du système de 0.63 pour chuter jusqu'à 20% avec une utilisation de 0.96. N'étant pas satisfaits du taux de succès, mais ayant déjà une base de travail pour ordonnancer le plus de VMs possibles dans un système, nous avons ensuite cherché un autre algorithme pour compléter l'ordonnancement des VMs qui n'ont pas pu soit avoir de paramètres harmoniques, soit respecter les échéances des tâches.

Pour cela, nous avons commencé par étudier les conditions d'ordonnançabilité des tâches au sein des VMs, ainsi que les paramètres des VMs, pour lesquels les tâches respectent leurs échéances. Nous avons également effectué des tests d'ordonnancement de VMs pour déterminer quel type d'ordonnancement fonctionne le mieux. Nous en avons déduit que plus la VM est exécutée régulièrement, plus les tâches ont tendance à respecter leurs échéances. À partir de cette déduction, nous avons pensé à un ordonnancement proche de P-fair.

Pour obtenir un tel ordonnancement, nous avions un pré-requis : nous devions obtenir un ordonnancement prenant en compte des VMs déjà ordonnancées, et l'utiliser de manière itérative. Ce pré-requis nous a ainsi obligés à trouver un ordonnancement pour les VMs, le plus proche de l'ordonnancement fluide harmonique, sans pouvoir s'appuyer sur les recherches déjà effectuées sur P-fair, celles-ci étant orientées multi-processeurs et en-ligne. Or nous avions besoin ici de connaître l'ordonnancement des VMs à l'avance et donc d'avoir un ordonnancement hors-ligne.

Pour cela, nous avons d'abord utilisé la programmation linéaire. Cette solution nous permet d'obtenir un résultat optimal au problème donné. Nous avons ainsi créé toutes les conditions nécessaires à la création du problème linéaire pour s'approcher le plus possible de l'ordonnancement donné par le modèle fluide utilisé par P-fair dont l'ordonnancement est basé sur le concept de lag. Cette solution aurait pu être utilisée, si celle-ci avait pu produire un résultat dans un temps raisonnable pour des systèmes tels que ceux utilisés dans le projet CEOS (avec une hyper-période de

potentiellement plusieurs milliers d'unités de temps). Mais le temps de calcul est prohibitif (au moins plusieurs mois).

Pour pallier cela, nous avons trouvé une autre solution : l'algorithme proportionnel itératif. Cet algorithme part ainsi d'un ordonnancement pour la VM, donné aléatoirement, et pour lequel les différents temps d'exécution de la VM seront modifiés pour s'approcher au mieux de l'ordonnancement du modèle fluide donné par P-fair.

Cette solution fonctionne d'autant mieux que nous avions vu qu'avec une utilisation inférieure à 0.9, environ 50% des systèmes avaient trouvé une solution pour exactement trois VMs. L'ordonnancement de la dernière VM devient trivial, puisqu'il n'y a plus qu'une VM à ordonnancer : il suffit alors d'ordonnancer cette dernière VM à chaque instant où aucune autre VM n'est exécutée. Cela nous permet ainsi d'éviter la phase de recherche pour s'approcher au mieux du lag en changeant les temps d'exécution de la VM étudiée dans d'autres endroits, où aucune autre VM n'est exécutée. L'algorithme obtenu est ainsi bien plus rapide pour trouver un ordonnancement s'approchant le plus possible, selon les possibilités laissées par les autres VMs, de l'ordonnancement du modèle fluide idéal de P-fair.

L'ajout d'un tel algorithme pour compléter l'ordonnancement harmonique nous a ainsi permis d'aller plus loin dans le taux de succès : cette fois, nous obtenons un taux de succès d'environ 60% pour une utilisation de 0.63 qui décroît jusqu'à 0% pour une utilisation de 1. Pour de tels systèmes, ce taux de succès pourrait être suffisant : il s'approche de 0% pour une utilisation de 0.88. Ce taux de succès correspond au seuil d'ordonnancement de RM. En effet, le seuil d'utilisation de RM en pratique sur des jeux de tâches aléatoires est fixé à 0.88 en moyenne (voir Section 2.2 du Chapitre 2). En revanche, ce taux reste moins bon avant ce seuil. Mais puisque nous nous intéressons à des systèmes utilisant un hyperviseur, ce taux de succès est meilleur mais n'est pas encore satisfaisant.

8.3 Ordonnancement des tâches

Nous avons par la suite souhaité aller plus loin pour obtenir un meilleur taux de succès. Ayant déjà obtenu un ordonnancement pour les VMs, nous avons ainsi cherché d'autres solutions pour ordonnancer les tâches au sein des VMs. Pour cela, nous avions d'abord envisagé EDF, mais cet algorithme pouvant être coûteux, nous avons opté pour Dual Priority. Cet algorithme n'est pas optimal, mais quasi optimal. Il est également moins coûteux à implémenter qu'EDF. Des Operating System (OS) tels que HIPPEROS commencent à essayer d'implémenter de telles solutions. C'est ce contexte qui nous a poussé à tester Dual Priority pour l'ordonnancement des tâches, en plus d'un ordonnancement à priorité fixe au niveau des tâches. La combinaison des deux a d'ailleurs un autre avantage : l'ordonnancement à priorité fixe au niveau des tâches est un cas particulier de Dual Priority, où l'échéance intermédiaire est égale à la période de la tâche. La combinaison des deux approches peut, au niveau de l'implémentation, ne pas nécessiter deux algorithmes différents.

Ce sont tous ces avantages qui nous ont poussé à tester Dual Priority pour l'ordonnancement des tâches. Nous avons d'abord commencé par étudier différents algorithmes de calcul des priorités, ainsi que de l'échéance de promotion pour Dual Priority, en contexte non hiérarchique, à savoir, FDMS et RM^{-1}/RM. Le premier utilise une simulation de l'ordonnancement Dual Priority, avec des paramètres de base, et lorsque une tâche ne respecte pas son échéance, son échéance intermédiaire

est alors décrémentée. Les simulations se font ainsi jusqu'à ce que le système soit ordonnançable ou qu'il ne soit plus possible de décrémenter l'échéance intermédiaire.

Cet algorithme est ainsi très efficace (proche de l'optimalité), mais très gourmand en ressources de calcul, puisqu'il peut potentiellement avoir un grand nombre de simulations (qui dépendent de l'hyper-période et des périodes des tâches). C'est pour cela que l'algorithme RM^{-1}/RM est également proposé. Son taux de succès est légèrement inférieur à FDMS, mais celui-ci permet de trouver les paramètres Dual Priority des tâches sans simulation, et avec un coût de calcul très faible.

Notre objectif suivant a été d'intégrer Dual Priority à notre modèle hiérarchique à deux niveaux. Pour cela, nous avons repris nos algorithmes précédents pour les VMs, à savoir l'algorithme harmonique combiné à l'algorithme proportionnel itératif. Cela nous permet ainsi d'avoir un ordonnancement pour les VMs. Dual Priority pouvant être coûteux, nous avons d'abord testé un ordonnancement à priorité fixe classique, et pour chaque VM ordonnancée, dont les tâches ne respectent pas leurs échéances, nous testons alors Dual Priority sur ce jeu de tâches grâce à l'algorithme FDMS.

Cela nous a ainsi permis d'obtenir les taux de succès finaux en combinant toutes les approches vues dans cette thèse. Ces résultats sont ainsi bien meilleurs : nous obtenons maintenant un taux de succès de plus de 90% pour une utilisation de 0.63 descendant progressivement jusqu'à 0% lorsque l'utilisation des tâches tend vers 1. La décroissance est proche d'une décroissance linéaire. Nous obtenons des résultats que nous estimons satisfaisants bien que nous ne puissions pas les comparer à d'autres résultats, puisque le modèle hiérarchique à deux niveaux présenté dans cette thèse n'a pas encore été utilisé, à notre connaissance, dans d'autres recherches.

8.4 Perspectives

Parmi ce que nous avons présenté précédemment, il existe des pistes pour améliorer encore les performances. Ainsi, nous pouvons principalement citer les approches P-fair et Dual Priority. En effet, concernant les approches P-fair, il est possible d'explorer davantage la programation linéaire. Par exemple, en expérimentant différents algorithmes pour identifier le plus efficace. Pour cela, il faudrait travailler en collaboration avec une équipe spécialisée dans la programmation linéaire.

En ce qui concerne la seconde approche, il serait souhaitable d'intégrer totalement les overheads dans l'algorithme proportionnel itératif. En effet, dans notre version, nous partons du principe que des VMs sont déjà ordonnancées, et que notre algorithme permettra d'obtenir des temps d'exécution de la VM supérieurs à une unité de temps. Aucune garantie n'est faite ici, mais pour pouvoir en obtenir, il faut que l'algorithme ne soit pas itératif mais puisse ordonnancer toutes les VMs en même temps, pour s'assurer que chaque exécution de VM tienne compte de son overhead. Ce comportement étant contraire à ce que nous cherchions, nous avons légitimement mis de côté cette partie. D'autant plus que, comme présenté précédemment, nous n'avions, la plupart du temps, qu'une ou deux VMs à ordonnancer avec cet algorithme.

En ce qui concerne l'approche harmonique, pour augmenter la probabilité d'obtenir des VMs ordonnançables, il faudrait envisager d'intégrer la Dual Priority dès l'ordonnancement de la première VM.

La dernière perspective envisagée concerne le développement logiciel effectué lors de cette thèse. En effet, dans notre domaine de recherche, chaque laboratoire, voire chaque chercheur développe son propre simulateur. Or, nous avons produit un

simulateur d'ordonnancement temps réel ainsi qu'un générateur de tâches. Celui-ci, se voulant, à la base, le plus générique possible, et étant écrit dans un langage largement utilisé (à savoir Java), il serait possible de compléter sa documentation et de le mettre en ligne en tant que logiciel libre. Cela pourrait grandement aider toutes les recherches futures dans ce domaine, en fournissant un seul outil. Chacun pourrait y contribuer et l'enrichir pour le spécialiser dans ses propres recherches. Les résultats de simulation seraient ainsi plus faciles à comparer, puisqu'un standard existerait. Il est également possible d'y ajouter une interface graphique.

8.5 Liste des publications

- Tristan Fautrel, Laurent George, Frédéric Fauberteau, Thierry Grandpierre, « An hypervisor approach for mixed critical real-time UAV applications ». Publié dans : *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 11-15 mars 2019, Kyoto, Japon.

- Tristan Fautrel, Laurent George, Joël Goossens, Damien Masson, Paul Rodriguez, « A Practical Sub-Optimal Solution for the Dual Priority Scheduling Problem ». Publié dans : *2018 IEEE 13th International Symposium on Industrial Embedded Systems (SIES)*, 6-8 juin 2018, Graz, Autriche.

- Tristan Fautrel, Laurent George, Frédéric Fauberteau, « An hypervisor schedulability analysis for safety and security critical applications scheduled in arbitrary patterns of slots ». Publié dans *Proc. of JRWRTC, 2017 - rtns17.org*, 4-6 octobre, Grenoble, France.

Bibliographie

[1] Tarek F ABDELZAHER et Kang G SHIN. « Combined task and message scheduling in distributed real-time systems ». In : *IEEE Transactions on parallel and distributed systems* 10.11 (1999), p. 1179-1191.

[2] Luca ABENI et Giorgio BUTTAZZO. « Integrating multimedia applications in hard real-time systems ». In : *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE. 1998, p. 4-13.

[3] Luís ALMEIDA. « Response time analysis and server design for hierarchical scheduling ». In : *proceedings of the IEEE Real-Time Systems Symposium Work-in-Progress*. Citeseer. 2003.

[4] James H ANDERSON et Anand SRINIVASAN. « Early-release fair scheduling ». In : *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*. IEEE. 2000, p. 35-43.

[5] Björn ANDERSSON, Sanjoy BARUAH et Jan JONSSON. « Static-priority scheduling on multiprocessors ». In : *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*. IEEE. 2001, p. 193-202.

[6] Björn ANDERSSON et Konstantinos BLETSAS. « Sporadic multiprocessor scheduling with few preemptions ». In : *2008 Euromicro Conference on Real-Time Systems*. IEEE. 2008, p. 243-252.

[7] Björn ANDERSSON et Jan JONSSON. « Fixed-priority preemptive multiprocessor scheduling : to partition or not to partition ». In : *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*. IEEE. 2000, p. 337-346.

[8] Björn ANDERSSON et Eduardo TOVAR. « Multiprocessor scheduling with few preemptions ». In : *12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'06)*. IEEE. 2006, p. 322-334.

[9] Dalia AOUN, Anne-Marie DÉPLANCHE et Yvon TRINQUET. « Pfair scheduling improvement to reduce interprocessor migrations ». In : *16th International Conference on Real-Time and Network Systems (RTNS 2008)*. 2008.

[10] Mikael ÅSBERG, Thomas NOLTE et Shinpei KATO. « Towards hierarchical scheduling in linux/multi-core platform ». In : *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*. IEEE. 2010, p. 1-4.

[11] Mikael ÅSBERG et al. « Towards real-time scheduling of virtual machines without kernel modifications ». In : *Emerging Technologies & Factory Automation (ETFA), 2011 IEEE 16th Conference on*. IEEE. 2011, p. 1-4.

[12] IEEE Standards ASSOCIATION et al. *802.11 a-1999—IEEE Standard for Telecommunications and Information Exchange Between Systems—LAN/MAN Specific Requirements—Part 11 : Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications : High Speed Physical Layer in the 5 GHz Band*. Rapp. tech. Technical Report, 1999.

[13] Neil C AUDSLEY. *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. Citeseer, 1991.

[14] Theodore P BAKER. « An analysis of EDF schedulability on a multiprocessor ». In : *IEEE transactions on parallel and distributed systems* 16.8 (2005), p. 760-768.

[15] Theodore P BAKER et Sanjoy K BARUAH. « Schedulability analysis of multiprocessor sporadic task systems ». In : *Handbook of Real-Time and Embedded Systems*. Chapman et Hall/CRC, 2007, p. 49-66.

[16] Sanjoy BARUAH. *Mixed criticality schedulability analysis is highly intractable*. 2009.

[17] Sanjoy BARUAH, Arvind EASWARAN et Zhishan GUO. « Mixed-criticality scheduling to minimize makespan ». In : (2016).

[18] Sanjoy BARUAH, Haohan LI et Leen STOUGIE. « Towards the design of certifiable mixed-criticality systems ». In : *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2010, p. 13-22.

[19] Sanjoy BARUAH et al. « Scheduling real-time mixed-criticality jobs ». In : *IEEE Transactions on Computers* 61.8 (2011), p. 1140-1152.

[20] Sanjoy K BARUAH et John CARPENTER. « Multiprocessor fixed-priority scheduling with restricted interprocessor migrations ». In : *Journal of Embedded Computing* 1.2 (2005), p. 169-178.

[21] Sanjoy K BARUAH, Aloysius K MOK et Louis E ROSIER. « Preemptively scheduling hard-real-time sporadic tasks on one processor ». In : *[1990] Proceedings 11th Real-Time Systems Symposium*. IEEE. 1990, p. 182-190.

[22] Sanjoy K BARUAH, Louis E ROSIER et Rodney R HOWELL. « Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor ». In : *Real-time systems* 2.4 (1990), p. 301-324.

[23] Sanjoy K BARUAH et al. « Proportionate progress : A notion of fairness in resource allocation ». In : *Algorithmica* 15.6 (1996), p. 600-625.

[24] SK BARUAH et Zhishan GUO. « Mixed criticality scheduling upon unreliable processors ». In : *University of North Carolina at Chapel Hill, Chapel Hill, NC, USA* (2013), p. 1-10.

[25] Jean-Luc BECHENNEC et al. « Trampoline an open source implementation of the osek/vdx rtos specification ». In : *2006 IEEE Conference on Emerging Technologies and Factory Automation*. IEEE. 2006, p. 62-69.

[26] Ron BELL. « Introduction to IEC 61508 ». In : *Proceedings of the 10th Australian workshop on Safety critical systems and software-Volume 55*. Australian Computer Society, Inc. 2006, p. 3-12.

[27] Enrico BINI et Giorgio C. BUTTAZZO. « Measuring the Performance of Schedulability Tests ». In : *Real-Time Systems* 30.1–2 (2005), p. 129-154.

[28] Konstantinos BLETSAS et Björn ANDERSSON. « Notional processors : an approach for multiprocessor scheduling ». In : *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2009, p. 3-12.

[29] Felix BRUNS et al. « An evaluation of microkernel-based virtualization for embedded real-time systems ». In : *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*. IEEE. 2010, p. 57-65.

[30] Almut BURCHARD et al. « New strategies for assigning real-time tasks to multiprocessor systems ». In : *IEEE transactions on computers* 44.12 (1995), p. 1429-1442.

[31] A BURNS et AJ WELLINGS. « A Practical Method for Increasing Processor Utilisation ». In : *Proceedings, 5th Euromicro Workshop on Real-Time Systems*. Citeseer. 1993.

[32] Alan BURNS. « Dual priority scheduling : Is the processor utilisation bound 100% ». In : *Proc. of the 1st International Real-Time Scheduling Open Problems Seminar (RTSOPS), in conjunction with the ECRTS*. 2010.

[33] Huacai CHEN et al. « Adaptive audio-aware scheduling in xen virtual environment ». In : *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on*. IEEE. 2010, p. 1-8.

[34] Ludmila CHERKASOVA, Diwaker GUPTA et Amin VAHDAT. « Comparison of the three CPU schedulers in Xen ». In : *SIGMETRICS Performance Evaluation Review* 35.2 (2007), p. 42-51.

[35] E COFFMAN. « Introduction to deterministic scheduling theory ». In : *Computer and Job-Shop Scheduling Theory* (1976), p. 1-50.

[36] IEEE Computer Society LAN MAN Standards COMMITTEE et al. « Wireless LAN medium access control (MAC) and physical layer (PHY) specifications ». In : *ANSI/IEEE Std. 802.11-1999* (1999).

[37] Tommaso CUCINOTTA, Gaetano ANASTASI et Luca ABENI. « Respecting temporal constraints in virtualised services ». In : *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*. T. 2. IEEE. 2009, p. 73-78.

[38] Tommaso CUCINOTTA et al. « Providing performance guarantees to virtual machines using real-time scheduling ». In : *European Conference on Parallel Processing*. Springer. 2010, p. 657-664.

[39] Sadegh DAVARI. « On a periodic real-time task allocation problem ». In : *Proc. of 19th Annual International Conference on System Sciences, 1986*. 1986, p. 133-141.

[40] Robert DAVIS et Andy WELLINGS. « Dual priority scheduling ». In : *Proceedings 16th IEEE Real-Time Systems Symposium*. IEEE. 1995, p. 100-109.

[41] Robert I DAVIS. *Dual priority scheduling : A means of providing flexibility in hard real-time systems*. Citeseer, 1994.

[42] Robert I DAVIS et Alan BURNS. « A survey of hard real-time scheduling for multiprocessor systems ». In : *ACM computing surveys (CSUR)* 43.4 (2011), p. 35.

[43] Robert I DAVIS et Alan BURNS. « Hierarchical fixed priority pre-emptive scheduling ». In : *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*. IEEE. 2005, 10-pp.

[44] Robert I DAVIS, Alan BURNS et W WALKER. « Guaranteeing timing constraints under shortest remaining processing time scheduling ». In : *Proceedings Ninth Euromicro Workshop on Real Time Systems*. IEEE. 1997, p. 88-93.

[45] Joe DECUIR et al. « Bluetooth 4.0 : low energy ». In : *Cambridge, UK : Cambridge Silicon Radio SR plc* 16 (2010).

[46] David DÉHARBE, Stephenson GALVAO et Anamaria Martins MOREIRA. « Formalizing freertos : First steps ». In : *Brazilian Symposium on Formal Methods*. Springer. 2009, p. 101-117.

[47] Matthew DELLINGER, Piyush GARYALI et Binoy RAVINDRAN. « ChronOS Linux : a best-effort real-time multiprocessor Linux kernel ». In : *Proceedings of the 48th Design Automation Conference*. ACM. 2011, p. 474-479.

[48] Zhong DENG et JW-S LIU. « Scheduling real-time applications in an open environment ». In : *Proceedings Real-Time Systems Symposium*. IEEE. 1997, p. 308-319.

[49] Michael DERTOUZOS. « Control Robotics : The Procedural Control of Physical Processeds ». In : *Proc. IFIP congress*. 1974, p. 807-813.

[50] Raymond DEVILLERS et Joël GOOSSENS. « Liu and Layland's schedulability test revisited ». In : *Information Processing Letters* 73.5-6 (2000), p. 157-161.

[51] Sudarshan K DHALL et Chung Laung LIU. « On a real-time scheduling problem ». In : *Operations research* 26.1 (1978), p. 127-140.

[52] Friedrich EISENBRAND et al. « Scheduling periodic tasks in a hard real-time environment ». In : *Automata, languages and programming* (2010), p. 299-311.

[53] Heiko EISSFELDT. « POSIX : a developer's view of standards ». In : *Proceedings of the annual conference on USENIX Annual Technical Conference*. USENIX Association. 1997, p. 24-24.

[54] Pontus EKBERG. « Dual Priority Scheduling is Not Optimal ». In : *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[55] Paul EMBERSON, Roger STAFFORD et Robert I DAVIS. « Techniques for the synthesis of multiprocessor tasksets ». In : *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*. 2010, p. 6-11.

[56] ERIKA ENTERPRISE. *Open-source RTOS OSEK/VDX kernel*. 2017.

[57] Nathan FISHER, Sanjoy BARUAH et Theodore P BAKER. « The partitioned scheduling of sporadic tasks according to static-priorities ». In : *18th Euromicro Conference on Real-Time Systems (ECRTS'06)*. IEEE. 2006, 10-pp.

[58] Nathan Wayne FISHER. *The multiprocessor real-time scheduling of general task systems*. Citeseer, 2007.

[59] John FOGELIN. « The VxWorks real-time kernel ». In : *Wind River Systems Inc* 1010 ().

[60] Gerhard FOHLER. « How different are offline and online scheduling ? » In : *RTSOPS 2011* (2011).

[61] Gerhard FOHLER et Krithi RAMAMRITHAM. « Static scheduling of pipelined periodic tasks in distributed real-time systems ». In : *Proceedings Ninth Euromicro Workshop on Real Time Systems*. IEEE. 1997, p. 128-135.

[62] Gerhard J FOHLER. *Flexibility in statically scheduled hard real-time systems*. Citeseer, 1994.

[63] Kenji FUNAOKA, Shinpei KATO et Nobuyuki YAMASAKI. « Energy-efficient optimal real-time scheduling on multiprocessors ». In : *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*. IEEE. 2008, p. 23-30.

[64] Shelby FUNK, Joel GOOSSENS et Sanjoy BARUAH. « On-line scheduling on uniform multiprocessors ». In : *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001)(Cat. No. 01PR1420)*. IEEE. 2001, p. 183-192.

[65] Simon FÜRST et Markus BECHTER. « AUTOSAR for connected and autonomous vehicles : The AUTOSAR adaptive platform ». In : *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE. 2016, p. 215-217.

[66] Laurent GEORGE, Pierre COURBIN et Yves SOREL. « Job vs. portioned partitioning for the earliest deadline first semi-partitioned scheduling ». In : *Journal of Systems Architecture* 57.5 (2011). Special Issue on Multiprocessor Real-time Scheduling, p. 518-535. ISSN : 1383-7621. DOI : https://doi.org/10.1016/j.sysarc.2011.02.008. URL : https://www.sciencedirect.com/science/article/pii/S1383762111000324.

[67] Laurent GEORGE, Joël GOOSSENS et Damien MASSON. « Dual Priority and EDF : a closer look ». In : 2014.

[68] Laurent GEORGE et Jean-Francois HERMANT. « A norm approach for the Partitioned EDF Scheduling of Sporadic Task Systems ». In : *2009 21st Euromicro Conference on Real-Time Systems*. IEEE. 2009, p. 161-169.

[69] Laurent GEORGE, Paul MUHLETHALER et Nicolas RIVIERRE. « Optimality and non-preemptive real-time scheduling revisited ». In : (1995).

[70] Laurent GEORGE, Nicolas RIVIERRE et Marco SPURI. *Preemptive and Non–Preemptive Real-Time UniProcessor Scheduling*. Rapp. tech. RR–2966. INRIA, 1996.

[71] Philippe GERUM. « Xenomai-Implementing a RTOS emulation framework on GNU/Linux ». In : *White Paper, Xenomai* (2004), p. 81.

[72] Teguh M GHAZALIE et Theodore P. BAKER. « Aperiodic servers in a deadline scheduling environment ». In : *Real-Time Systems* 9.1 (1995), p. 31-67.

[73] Joël GOOSSENS. « Scheduling of offset free systems ». In : *Real-Time Systems* 24.2 (2003), p. 239-258.

[74] Joël GOOSSENS, Shelby FUNK et Sanjoy BARUAH. « Priority-driven scheduling of periodic task systems on multiprocessors ». In : *Real-time systems* 25.2-3 (2003), p. 187-205.

[75] Joël GOOSSENS et Christophe MACQ. « Limitation of the Hyper–Period in Real-Time Periodic Task Set Generation ». In : *Proc. of RTS*. 2001, p. 133-148.

[76] Sriram GOVINDAN et al. « Xen and co. : Communication-aware cpu management in consolidated xen-based hosting platforms ». In : *IEEE Transactions on Computers* 58.8 (2009), p. 1111-1125.

[77] Mentor GRAPHICS. « Nucleus RTOS ». In : *URL, Accessed October* (2009).

[78] Flavius GRUIAN et Krzysztof KUCHCINSKI. « Uncertainty-based scheduling : energy-efficient ordering for tasks with variable execution time ». In : *Proceedings of the 2003 international symposium on Low power electronics and design*. ACM. 2003, p. 465-468.

[79] Chuancai GU et al. « Improving OCBP-based scheduling for mixed-criticality sporadic task systems ». In : *2013 IEEE 19th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2013, p. 247-256.

[80] Ana GUASQUE, Patricia BALBASTRE et Alfons CRESPO. « Real-time hierarchical systems with arbitrary scheduling at global level ». In : *J. Syst. Softw.* 119 (2016), p. 70-86.

[81] Zhishan GUO, Luca SANTINELLI et Kecheng YANG. « Mixed-Criticality Scheduling with Limited HI-Criticality Behaviors ». In : *International Symposium on Dependable Software Engineering : Theories, Tools, and Applications*. Springer. 2018, p. 187-199.

[82] Diwaker GUPTA et al. « Enforcing performance isolation across virtual machines in Xen ». In : *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer. 2006, p. 342-362.

[83] Sangchul HAN et Minkyu PARK. « Predictability of least laxity first scheduling algorithm on multiprocessor real-time systems ». In : *International Conference on Embedded and Ubiquitous Computing*. Springer. 2006, p. 755-764.

[84] Sanghyun HAN et Hyun-Wook JIN. « Full virtualization based ARINC 653 partitioning ». In : *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*. IEEE. 2011, 7E1-1.

[85] Martin HEUSSE et al. « Performance anomaly of 802.11 b ». In : *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*. T. 2. IEEE. 2003, p. 836-843.

[86] Jens HILDEBRANDT, Frank GOLATOWSKI et Dirk TIMMERMANN. « Scheduling coprocessor for enhanced least-laxity-first scheduling in hard real-time systems ». In : *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*. IEEE. 1999, p. 208-215.

[87] Philip L HOLMAN. *On the implementation of pfair-scheduled multiprocessor systems*. The University of North Carolina at Chapel Hill, 2004.

[88] Kwang Soo HONG et Joseph YT LEUNG. « Preemptive scheduling with release times and deadlines ». In : *Real-Time Systems* 1.3 (1989), p. 265-281.

[89] Kwang Soo HONG et JY-T LEUNG. « On-line scheduling of real-time tasks ». In : *IEEE transactions on Computers* 10 (1992), p. 1326-1331.

[90] Chao-Ju HOU et Kang G. SHIN. « Allocation of periodic task modules with precedence and deadline constraints in distributed real-time systems ». In : *IEEE transactions on computers* 46.12 (1997), p. 1338-1356.

[91] Asif IQBAL, Nayeema SADEQUE et Rafika Ida MUTIA. « An overview of microkernel, hypervisor and microvisor virtualization approaches for embedded systems ». In : *Report, Department of Electrical and Information Technology, Lund University, Sweden* 2110 (2009), p. 15.

[92] Leslie A JOHNSON et al. « DO-178B, Software considerations in airborne systems and equipment certification ». In : *Crosstalk, October* 199 (1998).

[93] Jan JONSSON et Kang G SHIN. « A parametrized branch-and-bound strategy for scheduling precedence-constrained tasks on a multiprocessor system ». In : *Proceedings of the 1997 International Conference on Parallel Processing (Cat. No. 97TB100162)*. IEEE. 1997, p. 158-165.

[94] Mathai JOSEPH et Paritosh PANDYA. « Finding response times in a real-time system ». In : *The Computer Journal* 29.5 (1986), p. 390-395.

[95] Jennifer Whitehead JOSEPH Y.-T. LEUNG. « On the complexity of fixed-priority scheduling of periodic, real-time tasks ». In : *Performance evaluation* 2 (1982), p. 237-250.

[96] WN JOY. « An introduction to the C shell. UNIX Programmer's Manual, 4.2 Berkeley Software Distribution. Computer Systems Research Group, Depat. of Electrical Engineering and Computer Science ». In : *Univ. of California, Berkeley* (1980).

[97] Robert KAISER et Stephan WAGNER. « Evolution of the PikeOS Microkernel ». In : *Proc. of MIKES*. 2007, p. 50-57.

[98] DI KATCHER, JP LEHOCZKY et JK STROSNIDER. « Scheduling models of dynamic priority schedulers ». In : *Research Report* (1993).

[99] Shinpei KATO et Nobuyuki YAMASAKI. « Portioned EDF-based scheduling on multiprocessors ». In : *Proceedings of the 8th ACM international conference on Embedded software*. ACM. 2008, p. 139-148.

[100] Shinpei KATO et Nobuyuki YAMASAKI. « Semi-partitioned fixed-priority scheduling on multiprocessors ». In : *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2009, p. 23-32.

[101] Shinpei KATO, Nobuyuki YAMASAKI et Yutaka ISHIKAWA. « Semi-partitioned scheduling of sporadic task systems on multiprocessors ». In : *2009 21st Euromicro Conference on Real-Time Systems*. IEEE. 2009, p. 249-258.

[102] Jan KISZKA et Bernardo WAGNER. « RTnet-a flexible hard real-time networking framework ». In : *2005 IEEE Conference on Emerging Technologies and Factory Automation*. T. 1. IEEE. 2005, 8-pp.

[103] Tei-Wei KUO et Ching-Hui LI. « A fixed-priority-driven open environment for real-time applications ». In : *Proc. of RTSS*. 1999, p. 256-267.

[104] Nandakishore KUSHALNAGAR et Gabriel MONTENEGRO. « Transmission of IPv6 packets over IEEE 802.15. 4 networks ». In : (2007).

[105] J LABETOULLE et al. « SOME THEOREMS ON REAL TIME SCHEDULING. » In : (1974).

[106] J LABROSSE. *MicroC/OS-III : The Real-Time Kernel, or a High Performance, Scalable, ROMable, Preemptive, Multitasking Kernel for Microprocessors, Microcontrollers & DSPs*. 2009.

[107] Jean J LABROSSE. « Microc/os-ii ». In : *R & D Books* 9 (1998).

[108] Vincent LEGOUT, Mathieu JAN et Laurent PAUTET. « Mixed-criticality multiprocessor real-time systems : Energy consumption vs deadline misses ». In : 2013.

[109] John LEHOCZKY, Lui SHA et Yuqin DING. « The rate monotonic scheduling algorithm : Exact characterization and average case behavior ». In : *[1989] Proceedings. Real-Time Systems Symposium*. IEEE. 1989, p. 166-171.

[110] John P LEHOCZKY. « Fixed priority scheduling of periodic task sets with arbitrary deadlines ». In : *Proc. of RTSS*. IEEE. 1990, p. 201-209.

[111] John P LEHOCZKY, Lui Raymond SHA et Jay K STROSNIDER. « Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. » In : *Unknown Host Publication Title*. IEEE, 1987, p. 261-270.

[112] J. Y. T. LEUNG et M.L. MERRIL. « A note on premptive scheduling of periodic, Real Time Tasks ». In : *Information Processing Letters.Vol 11* num 3 (1980).

[113] Joseph Y-T LEUNG et Jennifer WHITEHEAD. « On the complexity of fixed-priority scheduling of periodic, real-time tasks ». In : *Performance evaluation* 2.4 (1982), p. 237-250.

[114] Haohan LI et Sanjoy BARUAH. « An algorithm for scheduling certifiable mixed-criticality sporadic task systems ». In : *2010 31st IEEE Real-Time Systems Symposium*. IEEE. 2010, p. 183-192.

[115] Haohan LI et Sanjoy BARUAH. « Load-based schedulability analysis of certifiable mixed-criticality systems ». In : *Proceedings of the tenth ACM international conference on Embedded software*. ACM. 2010, p. 99-108.

[116] Giuseppe LIPARI et Enrico BINI. « A methodology for designing hierarchical scheduling systems ». In : *Journal of Embedded Computing* 1.2 (2005), p. 257-269.

[117] Chung Laung LIU et James W LAYLAND. « Scheduling algorithms for multiprogramming in a hard-real-time environment ». In : *Journal of the ACM (JACM)* 20.1 (1973), p. 46-61.

[118] Paolo MANTEGAZZA et al. « RTAI : Real-time application interface ». In : (2000).

[119] Miguel MASMANO et al. « Xtratum : a hypervisor for safety critical embedded systems ». In : *11th Real-Time Linux Workshop*. Citeseer. 2009, p. 263-272.

[120] Alejandro MASRUR et al. « Designing VM schedulers for embedded real-time applications ». In : *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM. 2011, p. 29-38.

[121] Alejandro MASRUR et al. « VM-based real-time services for automotive control applications ». In : *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*. IEEE. 2010, p. 218-223.

[122] D. MAXIM et L. CUCU-GROSJEAN. « Response Time Analysis for Fixed-Priority Tasks with Multiple Probabilistic Parameters ». In : *2013 IEEE 34th Real-Time Systems Symposium*. 2013, p. 224-235. DOI : 10.1109/RTSS.2013.30.

[123] Patrick MEUMEU YOMSI. « Prise en compte du coût exact de la préemption dans l'ordonnancement temps réel monoprocesseur avec contraintes multiples ». Thèse de doctorat dirigée par Yves Sorel, Université de Paris 11, 2009. Thèse de doct. 2009, 1 vol. (8-V-218 p.) URL : http://www.theses.fr/2009PA112080.

[124] Brent A MILLER et Chatschik BISDIKIAN. *Bluetooth revealed : the insider's guide to an open specification for global wireless communication*. Prentice Hall PTR, 2001.

[125] Aloysius Ka-Lau MOK. « Fundamental design problems of distributed systems for the hard-real-time environment ». Thèse de doct. Massachusetts Institute of Technology, 1983.

[126] Henry NEUGASS et al. « VxWorks : An interactive development environment and real-time kernel for GMicro ». In : *Proceedings Eighth TRON Project Symposium*. IEEE. 1991, p. 196-207.

[127] Sung-Heun OH et Seung-Min YANG. « A modified least-laxity-first scheduling algorithm for real-time tasks ». In : *Proceedings Fifth International Conference on Real-Time Computing Systems and Applications (Cat. No. 98EX236)*. IEEE. 1998, p. 31-36.

[128] Yingfeng OH et Sang H SON. « Allocating fixed-priority periodic tasks on multiprocessor systems ». In : *Real-Time Systems* 9.3 (1995), p. 207-239.

[129] Yingfeng OH et Sang H SON. « Tight performance bounds of heuristics for a real-time scheduling problem ». In : *Submitted for Publication* (1993).

[130] *Open Synergy COQOS*. URL : http://www.opensynergy.com/en/Products/COQOS/.

[131] Rob PALIN et al. « ISO 26262 safety cases : Compliance and assurance ». In : (2011).

[132] Taeju PARK et Soontae KIM. « Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems ». In : *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*. IEEE. 2011, p. 253-262.

[133] Cynthia A PHILLIPS et al. « Optimal time-critical scheduling via resource augmentation ». In : *Algorithmica* 32.2 (2002), p. 163-200.

[134] Andreas PLATSCHEK et Georg SCHIESSER. « Migrating a OSEK run-time environment to the OVERSEE platform ». In : *Proceedings of 13th Real-Time Linux Workshop*. 2011.

[135] Gerald J POPEK et Robert P GOLDBERG. « Formal requirements for virtualizable third generation architectures ». In : *Communications of the ACM* 17.7 (1974), p. 412-421.

[136] *Projet Céos*. URL : https://www.ceos-systems.com/fr/Projet-CEOS-Pour-L-Inspection-D-Ouvrages-Par-Drones.html.

[137] *Quick Start Guide National Instruments Real-Time Hypervisor*. URL : http://www.ni.com/pdf/manuals/375174b.pdf.

[138] Krithi RAMAMRITHAM. « Allocation and scheduling of precedence-related periodic tasks ». In : *IEEE Transactions on Parallel and Distributed Systems* 6.4 (1995), p. 412-420.

[139] *Real-Time Systems GmbH*. URL : https://www.real-time-systems.com/.

[140] FEDERICO REGHENZANI et GIUSEPPE MASSARI. « The Real-Time Linux Kernel : A Survey on PREEMPT_RT The Real-Time Linux Kernel : A Survey on PREEMPT_RT ». In : ().

[141] Leanna RIERSON. *Developing safety-critical software : a practical guide for aviation software and DO-178C compliance*. CRC Press, 2017.

[142] Ismael RIPOLL, Alfons CRESPO et Aloysius K MOK. « Improvement in feasibility testing for real-time tasks ». In : *Real-Time Systems* 11.1 (1996), p. 19-39.

[143] Wind RIVER. « VxWorks 653 for Integrated Modular Avionics ». In : *Wind River White Paper, Alameda* (2007).

[144] Saowanee SAEWONG et al. « Analysis of Hierar hical Fixed-Priority Scheduling. » In : *ECRTS*. T. 2. 2002, p. 173.

[145] Sartaj SAHNI. « Preemptive scheduling with due dates ». In : *Operations Research* 27.5 (1979), p. 925-934.

[146] Yasmina SEDDIK et Zdenek HANZÁLEK. « Match-up scheduling of mixed-criticality jobs : Maximizing the probability of jobs execution ». In : *European Journal of Operational Research* 262.1 (2017), p. 46-59.

[147] Inder M SINGH et Mitch BUNNELL. « LynxOS : UNIX Rewritten For Real-Time ». In : *Ik_ _* (1990), p. 27.

[148] Wayne E SMITH. « Various optimizers for single-stage production ». In : *Naval Research Logistics Quarterly* 3.1-2 (1956), p. 59-66.

[149] Dario SOCCI et al. « Mixed critical earliest deadline first ». In : *2013 25th Euromicro Conference on Real-Time Systems*. IEEE. 2013, p. 93-102.

[150] Dario SOCCI et al. « Time-triggered mixed-critical scheduler on single and multi-processor platforms ». In : *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. IEEE. 2015, p. 684-687.

[151] Brinkley SPRUNT. « Aperiodic task scheduling for real-time systems ». Thèse de doct. Citeseer, 1990.

[152] Marco SPURI. « Analysis of deadline scheduled real-time systems ». In : (1996).

[153] Marco SPURI et Giorgio BUTTAZZO. « Scheduling aperiodic tasks in dynamic priority systems ». In : *Real-Time Systems* 10.2 (1996), p. 179-210.

[154] Anand SRINIVASAN et Sanjoy BARUAH. « Deadline-based scheduling of periodic task systems on multiprocessors ». In : *Information processing letters* 84.2 (2002), p. 93-98.

[155] W Richard STEVENS et Gary R WRIGHT. *TCP/IP illustrated (vol. 2) : the implementation*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[156] Jay K. STROSNIDER, John P. LEHOCZKY et Lui SHA. « The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments ». In : *IEEE Transactions on Computers* 44.1 (1995), p. 73-91.

[157] Premysl SUCHA et Zdenek HANZÁLEK. « Scheduling with start time related deadlines ». In : *2004 IEEE International Conference on Robotics and Automation (IEEE Cat. No. 04CH37508)*. IEEE. 2004, p. 267-272.

[158] *Sysgo PikeOS*. URL : https://www.sysgo.com/.

[159] *Tenasys eVM for Windows*. URL : http://www.tenasys.com/products/evm-for-windows/.

[160] Ken TINDELL, Alan BURNS et Andy J. WELLINGS. « Analysis of hard real-time communications ». In : *Real-Time Systems* 9.2 (1995), p. 147-171.

[161] Ken W TINDELL, Alan BURNS et Andy J. WELLINGS. « An extendible approach for analyzing fixed priority hard real-time tasks ». In : *Real-Time Systems* 6.2 (1994), p. 133-151.

[162] Manohar VANGA et al. « Supporting low-latency, low-criticality tasks in a certified mixed-criticality OS ». In : *Proceedings of the 25th International Conference on Real-Time Networks and Systems*. ACM. 2017, p. 227-236.

[163] Dimitris VASSIS et al. « The IEEE 802.11 g standard for high data rate WLANs ». In : *IEEE network* 19.3 (2005), p. 21-26.

[164] S. VESTAL. « Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance ». In : *Proc. of RTSS*. 2007, p. 239-243.

[165] Yuxin WANG, Xiaogang WANG et He GUO. « An Optimized Scheduling Strategy Based on Task Type In Xen ». In : *Advances in Automation and Robotics, Vol. 2*. Springer, 2011, p. 515-522.

[166] *Wind River Virtualization Overview*. URL : https://www.windriver.com/products/operating-systems/virtualization/.

[167] Karim YAGHMOUR. « Adaptive domain environment for operating systems ». In : *Opersys inc* (2001).

[168] Nobuyuki YAMASAKI et al. « Real-time scheduling with task splitting on multiprocessors ». In : *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*. IEEE. 2007, p. 441-450.

[169] Peijie YU et al. « Real-time enhancement for Xen hypervisor ». In : *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*. IEEE. 2010, p. 23-30.

[170] Fengxiang ZHANG et Alan BURNS. « Schedulability analysis for real-time systems with EDF scheduling ». In : *IEEE Transactions on Computers* 58.9 (2009), p. 1250-1258.

[171] Fengxiang ZHANG et Alan BURNS. « Schedulability analysis of EDF-scheduled embedded real-time systems with resource sharing ». In : *ACM Transactions on Embedded Computing Systems (TECS)* 12.3 (2013), p. 67.

[172] Qin ZHENG et Kang G SHIN. « On the ability of establishing real-time channels in point-to-point packet-switched networks ». In : *IEEE Transactions on Communications* 42.234 (1994), p. 1096-1105.