HAL
open science

Environnement de développement d'applications multipériodiques sur plateforme multicoeur. La boîte à outil SchedMCore

Mikel Cordovilla

To cite this version:

Mikel Cordovilla. Environnement de développement d'applications multipériodiques sur plateforme multicoeur. La boîte à outil SchedMCore. Calcul parallèle, distribué et partagé [cs.DC]. Université Paul Sabatier - Toulouse III, 2012. Français. NNT : . tel-00720709

HAL Id: tel-00720709

https://theses.hal.science/tel-00720709

Submitted on 25 Jul 2012

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par **l'Institut Supérieur de l'Aéronautique et de l'Espace**

Spécialité : *Informatique*

Présentée et soutenue par

Mikel Cordovilla Mesonero

Le 2 Avril 2012

Environnement de développement d'applications multipériodiques sur plateforme multicœur. La boîte à outil SchedMCore

---

JURY

Président	Jean-Pierre Talpin	Directeur de Recherches, INRIA Rennes
Rapporteur	Emmanuel Grolleau	Professeur, ENSMA Poitiers
Rapporteur	Olivier H. Roux	Professeur, IRCCyN Nantes
Examinatrice	Liliana Cucu-Grosjean	Chargé de Recherche, INRIA Nancy-Grand Est
Examinateur	Pierre-Emmanuel Hladik	Maître de Conférences, INSA Toulouse
Directeur de thèse	Frédéric Boniol	Professeur, ONERA Toulouse
Encadrant de thèse	Eric Noulard	Ingénieur de Recherche, ONERA Toulouse
Encadrante de thèse	Claire Pagetti	Ingénieur de Recherche, ONERA Toulouse

---

École doctorale	:	**Mathématiques, Informatique et Télécommunications de Toulouse**
Unité de recherche	:	**Équipe d'accueil ISAE-ONERA MOIS**
Directeur de Thèse	:	**Frédéric Boniol**
Encadrants de Thèse	:	**Mme Claire Pagetti - M. Eric Noulard**

Remerciements

Me gustaría agradecer el excelente trabajo de supervisión realizado por Claire, Eric y Frédéric. Su disponibilidad, sus consejos y su apoyo han influido manifiestamente en la finalización y la calidad de esta tesis.

Me gustaría agradecer a todos los miembros del departamento DTIM de ONERA que me han acogido y me han espoleado hasta la conclusión de esta tesis. Igualmente, me gustaría agradecer a los miembros del tribunal el interés mostrado por este trabajo y el tiempo que han dedicado en él.

Dejando lo profesional de lado y pasando a un plano más personal, me gustaría agradecer :

- a los tarados mentales que han vivido conmigo y han soportado, día y noche durante tres años, a un lunático empedernido ;
- a los/as colegas, ¿que decir a los colegas... ? gracias por la paciencia mostrada. En vuestro lugar yo creo que hubiera dejado de llamaros en dos meses ;
- a los/as correctores, ¡venga chavales, gracias y buen trabajo ! ;
- a los/as doctorandos/as de Onera que han compartido alegrías y penas durante este periodo (para los que no han terminado todavía, ¡¡ánimo ! !) ;
- a la familia, por estar detrás mío y entender lo incomprensible.

Bueno... en definitiva, gracias a todos/as los/as que me han apoyado en esta aventura de tres largos años.

Table des matières

Premiere Page i

Table des matières v

Table des figures ix

Introduction xi

Partie I État de l'art 1

Chapitre 1 Langage PRELUDE 3

1.1 Définition du langage 3

1.2 Compilation 4

1.3 Exécution monoprocesseur 8

1.4 Résumé 9

Chapitre 2 Ordonnancement multiprocesseur 11

2.1 Modèle de la cible et des tâches 11

2.2 Politiques d'ordonnancement 13

2.2.1 Définitions 13

2.2.2 Politiques existantes pour architectures multiprocesseur 15

2.2.3 Précédences 19

2.2.4 Optimalité 20

2.3 Analyse d'ordonnançabilité 21

2.3.1 Deux approches : méthodes analytiques et méthodes de parcours exhaustif 22

2.3.2 Algorithmes existants de parcours exhaustif 26

2.3.3 Ordonnancement optimal hors-ligne 34

2.4 Résumé 36

Chapitre 3 Environnement d'exécution sur cible multicœur **37**
3.1 Objectif 37
3.2 Simulateurs 38
3.3 Exécution 41
3.3.1 Système d'exploitation temps réel multiprocesseur 41
3.3.2 Solution intermédiaire : Meta-scheduler 45
3.4 Résumé 46

Partie II Contributions **47**

Chapitre 4 Modélisation d'un système temps réel **49**
4.1 Encodage de l'évolution des configurations des tâches 49
4.1.1 Encodage des configurations des tâches 50
4.1.2 Représentation des contraintes de précédence 54
4.2 Analyse d'ordonnançabilité par parcours exhaustif 60
4.2.1 Encodage des politiques d'ordonnancement 60
4.2.2 Encodage de l'exploration 64
4.3 Génération de paramètres hors-ligne 69
4.3.1 Affection hors-ligne de priorité fixe 70
4.3.2 Recherche d'une solution optimale hors-ligne 73
4.4 Résumé 76

Chapitre 5 Exécutif SchedMCore Runner **77**
5.1 Architecture de SchedMCore Runner 77
5.1.1 Différents niveaux d'utilisation 77
5.1.2 Entités de SchedMCore Runner 78
5.1.3 Dynamicité et portabilité 81
5.2 Implantation de SchedMCore Runner 82
5.2.1 Implantation multithreadée 82
5.2.2 Phase d'initialisation 83
5.2.3 Phase d'exécution nominale 87
5.3 Gestion spécifique 91
5.3.1 Gestion du mode 3 91
5.3.2 Besoins spécifiques pour Prelude 93
5.4 Résumé 93

Chapitre 6 Expérimentations avec SchedMCore **95**
6.1 Petit manuel d'utilisation de SchedMCore 95
6.1.1 Installation 95
6.1.2 Format des fichiers d'entrée 96
6.1.3 Utilisation SchedMCore Converter 97
6.1.4 Utilisation SchedMCore Runner 98
6.2 Expérimentations avec SchedMCore Converter 101
6.2.1 Génération de tâches 101
6.2.2 Critères d'évaluation des mesures 102
6.2.3 Performances de l'analyse d'ordonnançabilité des politiques en-ligne . 103
6.2.4 Expérimentation de la génération de paramètres hors-ligne 114
6.3 Étude de cas 115
6.3.1 Spécification en Prelude 116
6.3.2 Choix d'une politique d'ordonnancement adaptée 117
6.3.3 Exécution réelle du FAS 118
6.4 Résumé 119

Conclusions et perspectives **121**

Bibliographie **125**

Table des figures

1 Architecture logicielle simplifiée du FAS . . . xi
2 Échange de données entre FDIR et GNC_US . . . xii
3 Framework PRELUDE-SCHEDMCORE . . . xiii

1.1 Comportement temporel du nœud sampling . . . 5
1.2 Précédences périodiques . . . 6
1.3 Communication de `vs` entre τ_2 et τ_1 . . . 7
1.4 Communication de `vs` entre τ_2 et τ_3 . . . 8

2.1 Modèle de tâche . . . 12
2.2 Illustration des 3 niveaux de migration . . . 13
2.3 Exécution globale vs. exécution partitionnée . . . 14
2.4 Illustration des 3 niveaux de priorité . . . 15
2.5 Classification de Carpenter . . . 15
2.6 Exécution en FP . . . 16
2.7 Exécution en gEDF . . . 16
2.8 Exécution en gLLF . . . 17
2.9 Exécution en LLREF . . . 18
2.10 Encodage avec Chetto en multiprocesseur . . . 20
2.11 Exemple Hong . . . 21
2.12 Exemple Hong . . . 21
2.13 Illustration d'une anomalie [Gra69] . . . 22
2.14 Illustration de l'instant critique . . . 23
2.15 Exemple de simulation . . . 24
2.16 Exemple de non prédictibilité . . . 26
2.17 Guan et al. modèles . . . 27
2.18 Modèle de David et al. (1) . . . 28
2.19 Automates de politiques . . . 29
2.20 Modélisation TPN de deux tâches sur un processeur . . . 29
2.21 Dépliage selon l'algorithme de Baker et Cirinei . . . 32

3.1 Description de l'architecture . . . 40
3.2 Diagramme de séquence des tâches . . . 40
3.3 Exécution avec un RTOS . . . 42
3.4 Exécution avec le Meta-scheduler . . . 45

4.1 Exécution sur deux processeurs et FP . . . 56
4.2 Exécution en LLREF . . . 63

4.3 Exécution en LLREF modifié 63
4.4 Exploration non déterministe 66
4.5 Exécution in UPPAAL 67
4.6 Execution in UPPAAL 69
4.7 Automate de recherche 72
4.8 Ensemble de tâches faisable mais non ordonnançable par une politique en-ligne . 74
4.9 Exécution hors-ligne nécessitant un temps creux 74
4.10 Recherche d'une séquence hors-ligne : cas synchrone pour 2 processeurs 75
4.11 Recherche d'une séquence hors-ligne : cas asynchrone pour 2 processeurs 76

5.1 Plateforme d'exécution SCHEDMCORE RUNNER 77
5.2 Architecture de SCHEDMCORE RUNNER 78
5.3 Ordonnanceur$_i$ greffé dans le squelette 79
5.4 Fonction F() greffée dans une tâche 80
5.5 Séquence des interactions entre les entités 80
5.6 Création de threads dans le système 83
5.7 Initialisation simultanée à l'aide de deux barrières 84
5.8 Démarrage de SCHEDMCORE RUNNER 85
5.9 Gestion du paramètre `task_state` par un ordonnanceur en-ligne 90
5.10 Gestion du paramètre `task_state` par un ordonnanceur hors-ligne 91
5.11 Isolement de l'exécution des tâches temps réel 92

6.1 Analyse d'ordonnançabilité en-ligne 97
6.2 Chargement d'un ordonnancement hors-ligne avec SCHEDMCORE TRACER . . . 100
6.3 Ratio d'acceptation en C avec variation du nombre de tâches 104
6.4 Ratio d'acceptation en UPPAAL avec variation du nombre de tâches 105
6.5 Temps de vérification avec variation du nombre de tâches 106
6.6 Ratio d'acceptation avec variation de la charge processeur 107
6.7 Temps de vérification avec variation de la charge processeur 108
6.8 Ratio d'acceptation en C avec variation du nombre de tâche et de précédences . . 109
6.9 Ratio d'acceptation en UPPAAL avec variation du nombre de tâche et de précédences 110
6.10 Temps de calcul avec variation du nombre de processeurs 111
6.11 Temps de calcul avec variation de l'hyper-période (en C uniquement) 112
6.12 Temps de calcul version optimisée avec variation du nombre de tâches 113
6.13 Affectation statique de priorités hors-ligne 114
6.14 Affectation statique de priorités hors-ligne par heuristique 115
6.15 Calcul d'une séquence hors-ligne 116

Introduction

Contexte

Un *système embarqué* temps réel de *contrôle-commande* contrôle le comportement d'un système exposé à son environnement. Pour cela, il reçoit des informations de l'environnement à travers des capteurs, réalise les algorithmes de contrôle et agit sur le système grâce à des actionneurs. On parle également de *système réactif*. Ces systèmes de contrôle-commande sont, dans la plupart des cas, hautement critiques et, en conséquence, aucune défaillance catastrophique n'est permise. Les commandes de vol d'un aéronef sont un exemple classique de systèmes de contrôle-commande : les capteurs fournissent des informations telles que la pression ou l'altitude et les actionneurs physiques, appelés surfaces, agissent notamment sur les ailerons et les moteurs. L'exposition à l'environnement étant permanente, les trois actions (acquisition, contrôle et réaction) se réalisent en boucle et à une certaine fréquence afin de réguler le système au cours du temps. Le choix de la fréquence se fait de sorte que les algorithmes de contrôle satisfassent leurs conditions de stabilité et sont généralement en lien avec la fréquence d'arrivée des entrées.

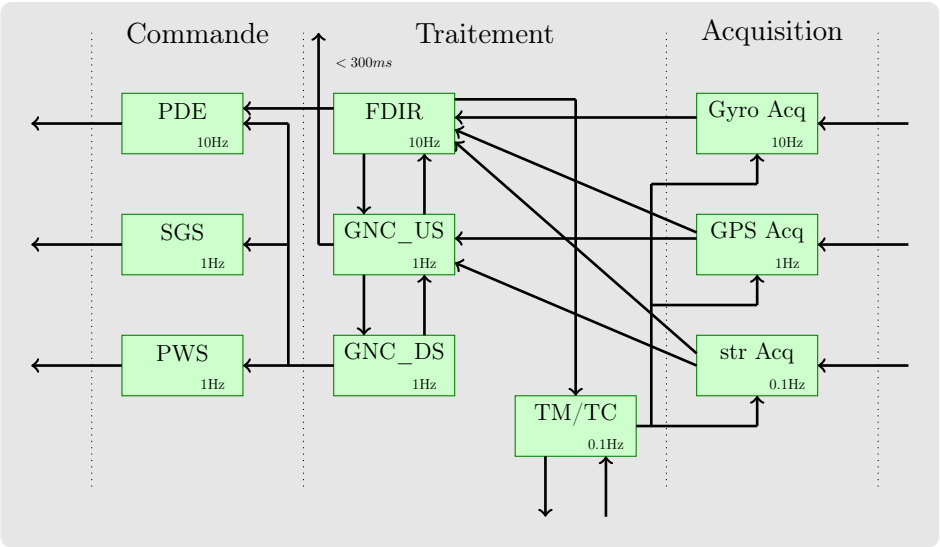


FIGURE 1 – Architecture logicielle simplifiée du FAS

Un deuxième exemple est le logiciel de navigation de l'Automatic Transfer Vehicle (ATV), véhicule spatial de ravitaillement de l'International Space Station (ISS). Considérons l'architecture logicielle simplifiée du *Flight Application Software*(FAS), dans la figure 1, développée par EADS Astrium Space Transportation. Cet exemple est issu de la thèse de Julien Forget [For09].

Le FAS est composé de plusieurs fonctions, représentées par des boîtes, dont les communications de données sont symbolisées par des flèches. Les flèches provenant ou se dirigeant vers l'extérieur représentent les entrées/sorties du système. Les fonctions sont classifiées en trois catégories (correspondant aux trois types d'action mentionnés précédemment). L'étape d'acquisition capte les données en entrée : l'orientation et la vitesse à partir des capteurs gyroscopiques (`Gyro Acq`), la position à partir du GPS (`GPS Acq`) et du star tracker (`str Acq`), enfin les télécommandes en provenance de la station sol (`TM/TC`). Le traitement consiste à calculer un certain nombre d'informations comme l'attitude, la position et l'avancement de la mission en fonction de l'état du véhicule. La fonction *Guidance Navigation and Control* (décomposée en deux sous-fonctions, `GNC_US`, et `GNC_DS`) calcule les commandes à appliquer. La fonction *Failure Detection, Isolation and Recovery* (`FDIR`) vérifie l'absence de pannes du FAS. La dernière étape est responsable de consolider les commandes et de les envoyer aux équipements : les commandes propulseurs pour le *Propulsion Drive Electronics* (`PDE`), les commandes de distribution d'énergie pour le *Power System* (`PWS`), les commandes de positionnement des panneaux solaires pour le *Solar Generation System* (`SGS`) et les télémétries à destination de la station sol (`TM/TC`).

Une des particularités de ces systèmes est la multipériodicité des fonctions (ou tâches) : le rythme d'exécution des tâches est imposé par des contraintes physiques, comme la vitesse d'acquisition d'un capteur, ou des règles de l'automatique, c'est pourquoi le rythme n'est pas forcément le même pour tous. Par exemple, le FAS est composé de tâches à trois rythmes différents : 10Hz, 1Hz et 0.1Hz. Cette caractéristique n'empêche pas l'échange de données entre des tâches de rythmes différents comme on peut le voir sur l'image 2. Dans ce schéma, `GNC_US` à 1hz et `FDIR` à 10hz communiquent entre elles. Étant donnée que la tâche `GNC_US` s'exécute plus rapidement, elle accomplit 10 exécutions pendant que la tâche `FDIR` n'en exécute qu'une seule. La spécification précise que `GNC_US` communique avec `FDIR` à la fin de la première exécution et `FDIR` communique avec la 3ème exécution de `GNC_US`.

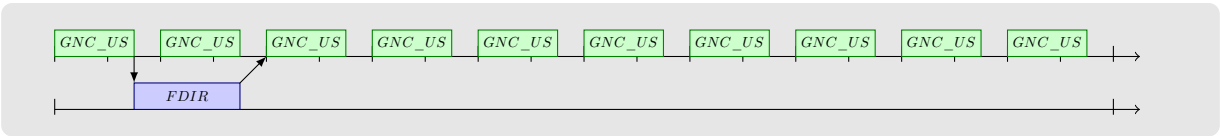


FIGURE 2 – Échange de données entre FDIR et GNC_US

La description de système contrôle-commande se fait à l'aide de langages de spécification de haut niveau. Dans cette thèse nous avons choisi le langage formel PRELUDE [For09] développé pour la spécification d'architecture logicielle de systèmes embarqués critiques. PRELUDE fait partie des langages appelés *synchrones* et se focalise essentiellement sur les aspects temps réel des systèmes multipériodiques. La sortie du compilateur PRELUDE est composée d'un ensemble de tâches temps réel avec des contraintes de précédences. Les communications multipériodiques sont gérées grâce à un protocole, dérivé de [STC06] et détaillé dans [For09]. Ce protocole est capable de garantir le respect des contraintes de précédences à l'exécution et le respect de la sémantique de la spécification.

Enfin, le compilateur génère du code et le système s'exécute sur une architecture monoprocesseur disposant d'un système d'exploitation temps réel, MarteOS [RH02]. En utilisant l'extension POSIX temps réel, les tâches sont codées comme des threads ordonnançables avec deux ordonnanceurs : Rate Monotonic (RM) et Earliest Deadline First (EDF).

L'évolution des systèmes contrôle-commande conduit vers une augmentation des fonctionnalités, en conséquence une augmentation de la puissance de calcul est requise. D'autre part, les architectures parallèles, notamment les multicœurs, sont de plus en plus présentes dans les

systèmes embarqués et offrent une augmentation importante de performance. Le partage de l'exécution sur ces nouvelles architectures pose de nouvelles difficultés à traiter pour une exécution correcte.

L'objectif de cette thèse est, à partir d'une spécification de haut niveau PRELUDE d'un système multipériodique, de générer un code multithreadé exécutable sur une architecture multicœur tout en respectant la sémantique initiale.

Contributions

La figure 3 illustre toutes les étapes nécessaires à l'exécution d'une spécification PRELUDE sur une cible multicœur. La partie supérieure décrit la démarche d'une spécification PRELUDE des nœuds importés jusqu'à la compilation en tâches temps réel. Cette partie correspond au travail de Julien Forget et est utilisée comme entrée par la boîte à outils SCHEDMCORE développée dans cette thèse. La partie *analyse d'ordonnançabilité multiprocesseur* est la première contribution de cette thèse. Elle prend en entrée des tâches temps réel (le format d'entrée est générique, la spécification initiale n'est pas nécessairement exprimée en PRELUDE) et propose d'une part une analyse d'ordonnançabilité et d'autre part la génération d'ordonnancement hors-ligne valide sur une architecture multiprocesseur. Une fois que la correction de l'exécution est vérifiée, l'objectif de la deuxième partie est l'*exécution sur une architecture cible*. Cette partie est la deuxième contribution de la thèse et permet trois modes d'exécutions correctes sur une cible multicœur respectant la sémantique initiale.

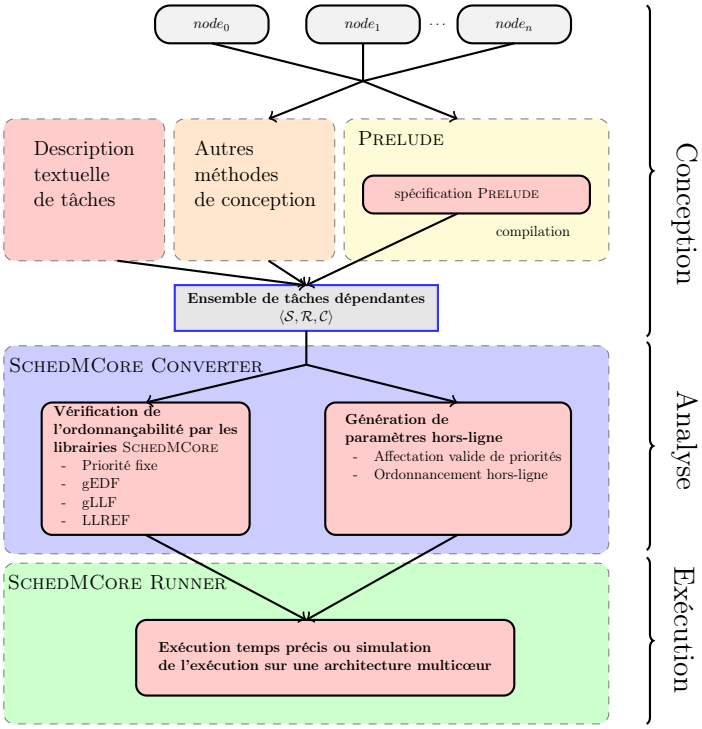


FIGURE 3 – Framework PRELUDE-SCHEDMCORE

Méthode exacte d'analyse d'ordonnançabilité

Une spécification PRELUDE génère un ensemble de tâches temps réel avec des contraintes de précédence et des motifs de communication précis. Le modèle généré par PRELUDE est de la forme $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$ où $\mathcal{S}$ est un ensemble de tâches caractérisées par quatre paramètres (O, T, D, C). O est la date de la première arrivée de la tâche au système, T est la période, D est l'échéance à laquelle la tâche doit finir son exécution et C est l'estimation du pire temps d'exécution. L'ensemble est aussi contraint par un certain nombre de précédences $\mathcal{R}$ et les communications sont représentées par $\mathcal{C}$. Un ensemble de tâches dont toutes les dates d'arrivée O sont nulles est dit *synchrone* et *asynchrone* sinon.

Pour exécuter un ensemble de tâches sur une cible physique, on utilise généralement une politique d'ordonnancement, c'est-à-dire une règle ou un algorithme qui décide quelle(s) tâche(s) s'exécute(nt) sur un processeur à un instant donné. Il existe plusieurs politiques dans la littérature pour ordonnancer des tâches sur une cible multiprocesseur (pour plus d'informations, le lecteur peut regarder [DB09]). Dans le cadre de cette thèse, nous nous sommes concentrés sur 5 politiques particulières. Les deux premières sont issues des travaux de Liu et Layland [LL73]. FP (Fixed Priority ou à priorité fixe) affecte une priorité constante aux tâches et gEDF (Global Earliest Deadline First) affecte une priorité constante à chaque répétition de tâches en fonction de l'échéance. La troisième gLLF (Global Least Laxity First) est une politique proposée par Mok [Mok83] qui se base sur la notion de laxité pour affecter les priorités. La laxité est la relation entre le temps restant jusqu'à la fin de l'échéance et la durée d'exécution restante à accomplir à cet instant. Enfin, une politique plus récente LLREF (Largest Local Remaining Execution First) [CRJ06] est basée à la fois sur l'idée d'une progression constante des tâches, introduite par Baruah [BGP95], et sur LLF. Cette politique divise l'exécution en quantums de temps dans lesquels deux situations peuvent arriver : soit une tâche a une laxité zéro, en conséquence elle doit nécessairement s'exécuter pour respecter ses contraintes, soit une tâche a une laxité strictement positive et alors elle respecte une exécution constante. Enfin, nous nous sommes intéressés également aux politiques hors-ligne qui consistent en l'obtention d'une trace d'exécution finie avant exécution du système.

Le problème de l'analyse d'ordonnançabilité vise à garantir formellement l'exécution correcte d'un système. Plus précisément, une analyse d'ordonnançabilité consiste à vérifier que, pour un ensemble de tâches, une politique et une cible donnés, toutes les exécutions possibles respectant cette politique sont valides, c'est-à-dire qu'elles respecteront toutes les contraintes temporelles et de précédence. L'analyse d'ordonnançabilité n'est pas concernée par les communications $\mathcal{C}$. Il existe deux approches pour analyser l'ordonnançabilité d'un ensemble de tâches : les méthodes analytiques ou les méthodes de parcours exhaustif. Nous avons choisi la deuxième solution car les tests exacts actuels ne sont pas toujours "satisfaisants" et ne traitent pas encore les précédences.

Le problème des méthodes de parcours exhaustif est celui de la cyclicité ou de la finitude de l'espace d'état. En effet, dans le cas monoprocesseur, la cyclicité est connue [CDKM02] et correspond à $[0, H]$, où H est le plus petit commun multiple (ppcm) de toutes les périodes des tâches, pour les ensembles synchrones à échéance contrainte. Pour les ensembles asynchrones à échéance contrainte, la fenêtre de répétition est $[0, max_i(O_i) + 2H]$. Dans le cas multiprocesseur Cucu et Goossens ont prouvé [CG07] que la cyclicité d'un ensemble de tâches synchrones à échéances contraintes est $[0, H]$. Pour les tâches asynchrones, ces auteurs ont également montré que l'exécution se répétait à partir d'un certain instant.

Pour contrer le problème de la cyclicité, nous avons utilisé un model checker et des algorithmes *ad hoc* (avec des idées proches de celles du model checking). Nous avons codé le problème de l'ordonnançabilité par un automate fini de configurations dont on doit explorer l'espace d'état.

Si un état particulier est (ou n'est pas, selon les cas) atteignable, alors l'ensemble de tâches est ordonnançable.

Trois types de problèmes ont été traités, comme exposé dans la figure 3. Le premier problème concerne la vérification de l'ordonnançabilité d'un ensemble de tâches pour une politique d'ordonnancement choisie par l'utilisateur (FP, gEDF, gLLF ou LLREF) et pour un certain nombre de processeurs. Le problème consiste alors à décider si l'ensemble de tâches respecte toutes les contraintes temporelles et de précédence (ces travaux sont présentés dans [CBNP11]). Le deuxième type de problème cherche, pour un ordonnancement FP, s'il existe une affectation de priorités qui respecte toutes les contraintes temporelles et de précédence. Le dernier problème cherche, une exécution (séquence) hors-ligne qui respecte toutes les contraintes, à la fois temporelles et de précédence (ces travaux sont décrits dans [CBNP12]).

Pour faciliter les analyses, nous avons développé une application SchedMCore Converter, qui transforme un modèle de tâche de type $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$ en des modèles Uppaal ou C représentant le problème d'ordonnancement à analyser. L'utilisateur exécute ensuite le programme C ou fait appel au model checker Uppaal pour décider l'ordonnançabilité ou calculer les solutions hors-ligne.

Exécutif

Une fois l'ordonnançabilité vérifiée, il faut exécuter l'ensemble de tâches sur la cible en respectant les contraintes. Nous supposons dans cette thèse que la question des motifs de communication $\mathcal{C}$ doit être gérée par l'exécutif. Dans sa thèse, J. Forget a proposé un protocole de communication à base de buffers en mémoire partagée. L'idée est de générer automatiquement des buffers entre deux tâches communicantes de façon à stocker toutes les valeurs nécessaires. Le compilateur Prelude génère également les moments de lecture et d'écriture dans les cases. Nous avons démontré chapitre 5.3.2 que le protocole de communication de Julien Forget s'applique dans le cas multiprocesseur. Nous avons contribué également à proposer des optimisations afin de réduire le nombre de buffers [PFB+11].

Nous avons ensuite cherché un environnement exécutif permettant de se rapprocher le plus fidèlement possible de l'exécution temps réel opérationnelle en simplifiant les aspects qui ne nous intéressaient pas notamment les entrée/sorties et les spécificités d'une cible particulière. Dans la littérature on trouve deux types d'approche pour exécuter un ensemble de tâches sur une architecture multiprocesseur : le système d'exploitation temps réel (RTOS pour real-time operating system) supportant l'exécution multicœur ou les implantations au niveau utilisateur. Le premier type de solution est complètement dédié à l'exécution temps réel et en conséquence, les couches les plus basses sont programmées dans ce but. Cette approche rend difficile les modifications ou les adaptations ainsi que la compatibilité avec l'architecture en requérant souvent la modification du noyau du système d'exploitation.

L'approche au niveau utilisateur se positionne au dessus du système d'exploitation et interagit avec lui au lieu de le modifier. Cette interaction est possible grâce à l'utilisation d'un standard comme Posix sur les systèmes compatibles ce qui est le cas de la majorité des (RT)OS et en particulier les dérivés de Linux. En revanche, la précision temporelle est moins bonne du fait de l'introduction des surcoûts liés au système qui sont difficilement calculables. Nous avons néanmoins choisi cette solution car l'objectif expérimental de ces travaux est de montrer la faisabilité, le prototypage rapide, ainsi que de permettre la validation fonctionnelle et temporelle de l'application utilisateur en acceptant un surcoût système qui pourrait être important, nécessitant une prise de marge suffisante au niveau utilisateur.

La plateforme SchedMCore Runner développée au cours de cette thèse permet d'exécuter

un ensemble de tâches sur une cible multicœur à mémoire partagée classique. Ce framework offre un ensemble de fonctionnalités facilitant l'exécution de tâches dans les systèmes basés sur Posix ou ayant quelques primitives systèmes équivalentes (nous verrons dans la partie 5.1.3 les mécanismes minimaux). L'utilisateur peut exécuter son ensemble de tâches avec une des politiques d'ordonnancement fournie, ou bien implémenter sa propre politique.

Dans le cas de l'obtention d'une séquence calculée hors-ligne, l'outil SchedMCore Tracer importe l'ordonnancement hors-ligne calculé par une librairie d'entrée dans SchedMCore Runner. L'exécution sera guidée à chaque instant par la séquence. L'environnement SchedMCore est décrit dans [CBF+11].

Plan du manuscrit

La première partie de ce mémoire rappelle l'état de l'art des différentes composantes de cette thèse. Ainsi, le chapitre 1 présente succinctement le langage d'entrée Prelude. Le chapitre 2 introduit certaines notions et résultats sur l'analyse d'ordonnançabilité multiprocesseur et le dernier chapitre 3 présente plusieurs exécutifs multiprocesseur existants.

La deuxième partie du manuscrit décrit les contributions de cette thèse. Dans le chapitre 4 nous présentons les résultats obtenus sur l'analyse d'ordonnançabilité. Tout d'abord, nous expliquons la modélisation du système, c'est-à-dire, l'encodage des tâches et la progression temporelle (section 4.1). Dans la section 4.2 nous présentons l'analyse d'ordonnançabilité pour des politiques en-ligne. La section 4.3 décrit la recherche de paramètres hors-ligne : la recherche d'affectation de priorité valide pour un ordonnanceur FP et la recherche d'une séquence ordonnançable.

Le chapitre 5 introduit le framework d'exécution SchedMCore Runner. Dans la partie 5.1, nous décrivons les choix de conception de l'exécutif. Dans la section 5.2, nous expliquons l'implantation du SchedMCore Runner et les particularités de l'exécution d'un ordonnanceur en-ligne et un ordonnanceur hors-ligne.

Le chapitre 6 s'intéresse aux aspects plus pratiques de la plateforme SchedMCore. Ainsi, dans la partie 6.1, nous présentons brièvement un manuel d'utilisation de la plateforme. Ensuite, dans la partie 6.2, nous décrivons des séries d'expérimentations menées avec l'outil SchedMCore Converter dans le but d'en évaluer les performances. Pour cela, nous avons également développé un générateur d'ensembles de tâches multipériodiques. Enfin, dans la partie 6.3, nous illustrons l'utilisation de l'environnement de conception SchedMCore en traitant l'exemple FAS introduit dans la figure 1.

Première partie

État de l'art

Chapitre 1

Langage PRELUDE

L'objectif de ce premier chapitre est de présenter brièvement le langage PRELUDE [FBLP08, For09] et les ensembles de tâches temps réel produits par le compilateur. Ce langage formel est dédié à la spécification d'architecture logicielle des systèmes de contrôle-commande multipériodiques.

Un programme PRELUDE prend en entrée un ensemble de nœuds importés (des fonctions décrites par l'utilisateur dans un autre formalisme). La contrainte imposée par PRELUDE est qu'une fonction importée est "synchrone" c'est-à-dire que les sorties sont calculées à partir des entrées au même rythme que celles-ci. On peut également voir la fonction comme un code séquentiel. Un programme PRELUDE décrit l'interaction temps réel entre les nœuds ainsi que les communications. Une fois le programme compilé, la sortie est composée d'un ensemble de tâches temps réel avec des contraintes de précédences et des motifs de communication.

1.1 Définition du langage

Les flots et les horloges

PRELUDE est un langage synchrone flot de données inspiré de LUSTRE [HCRP91], SIGNAL [BLGJ91] et LUCID SYNCHRONE [Pou06]. Toutes les variables et les expressions utilisées dans un programme sont en réalité des *flots* de données. Un flot est une séquence de couples $(v_i, t_i)_{i\in\mathbb{N}}$, où v_i est une valeur dans un domaine $\mathcal{V}$ et t_i est la date dans $\mathbb{Q}$ ($\forall t_i < t_{i+1}$). L'*horloge* d'un flot correspond à la projection sur $\mathbb{Q}$. Elle représente l'ensemble des instants pendant lesquels le flot a une valeur : v_i doit être produit pendant l'intervalle $[t_i, t_{i+1}[$. Cette hypothèse est appelée "synchrone retardée" [Cur05].

Deux flots sont *synchrones* entre eux s'ils ont la même horloge. Nous nous intéressons notamment à une classe spécifique d'horloges qui correspondent aux activations de tâches périodiques dénommées *horloges strictement périodiques* : l'horloge $h = (t_i)_{i\in\mathbb{N}}$ est strictement périodique s'il existe un rationnel n tel que $t_{i+1} - t_i = n$ pour tout i. n est la *période* de h et t_0 est la *phase*.

Hiérarchie de nœuds

Un programme PRELUDE est composé d'un ensemble de *nœuds*. Un nœud est défini par ses flots de sortie en fonction de ses flots d'entrée. Il y a trois types de nœuds : les nœuds importés écrits par l'utilisateur dans un autre langage, les capteurs ou actionneurs qui représentent des nœuds d'entrée/sortie et les nœuds PRELUDE (c'est un langage hiérarchique). Un programme est

ensuite constitué d'un ensemble d'équations définies pour exprimer la relation entre les entrées et les sorties. Un exemple de programme est donné ci-dessous :

```
imported node plus1(i: int) returns (o:int) wcet 5;

node N(i:int rate (10,0)) returns (o:int rate (10,0))
let o=plus1(i); tel
```

Dans ce premier programme, il y a un unique nœud importé `plus1`. Ce nœud a une unique entrée et produit une unique sortie, les deux étant des entiers. Le pire temps d'exécution de `plus1` sur la cible est de 5 unités de temps. Le nœud PRELUDE `N` décrit les interactions des entrées/sorties avec l'environnement. Selon le paradigme de programmation flot de données, le nœud `N` est actif à chaque instant où il reçoit une entrée `i`. Dans l'exemple il est actif avec une période de 10 unités de temps (le terme (10,0) indique l'horloge de période 10 et de phase 0). A chaque activation le système calcule le résultat de l'application du nœud importé `plus1` à `i`.

Opérations de transition de rythme

PRELUDE définit un ensemble d'*opérations de transition de rythme* qui permettent à l'utilisateur de décrire le motif de communication de données entre les nœuds de rythme différent. Soit e un flot de données, $k \in \mathbb{N}$ et $q \in \mathbb{Q}$:

- *cst* `fby` e est l'opérateur de délai similaire à celui de LUCID SYNCHRONE. D'abord il produit *cst* pour ensuite produire chaque valeur de e décalée d'un instant. L'horloge de ce flot est la même que celle de e ;
- $e *\hat{} k$ produit un flot dont la période est k fois plus courte que celle de e. Chaque valeur de e est doublée k fois dans le résultat.
- $e /\hat{} k$ produit un flot k fois plus lent que e. Le résultat produit la valeur de e toutes les k.
- $e \sim> q$ produit un flot où chaque valeur de e est retardée de $q * n$, où n est la période de e.

Ces opérateurs sont illustrés dans l'exemple suivant :

```
imported node tau_1 (i0 :int, i1: int) returns (o1 :int, o2 :int) wcet 5;
imported node tau_2 (i0 :int) returns (o1 :int) wcet 10;
imported node tau_3 (i0 :int, i1: int) returns (o1 :int) wcet 20;

node sampling (i : rate (10,0)) returns (o1, o2)
var vf, vs;
let
  (o1,vf)=tau_1(i, (0 fby vs)*^3);
  vs=tau_2(vf/^3);
  o2 = tau_3((vf~>1/10)/^6,(vs~>1/30)/^2);
tel
```

Dans ce deuxième exemple, 3 nœuds sont importés τ_i avec $i = 1, 2, 3$. Pour décrire les équations, deux variables locales `vf` et `vs` sont utilisées. Trois équations décrivent les relations entre les nœuds. Le comportement temporel de ce système est montré dans la figure 1.1 (nous détaillons les valeurs produites avec les dates de production).

1.2 Compilation

Le compilateur traduit un programme PRELUDE en un ensemble de tâches périodiques reliées par des contraintes de précédences et communicant selon des motifs de communication précis. La préservation de la sémantique du programme à travers les diverses étapes de compilation a été

date	0	1	10	20	30	40	50	60	61	...
`vf`	v_0		v_1	v_2	v_3	v_4	v_5	v_6		...
`vf/^3`	v_0				v_3			v_6		...
`vs`	v'_0				v'_1			v'_2		...
`0 fby vs`	0				v'_0			v'_1		...
`(0 fby vs)*^3`	0		0	0	v'_0	v'_0	v'_0	v'_1		...
`vf~>1/10/^6`		v_0							v_6	...
`vs~>1/30/^2`		v'_0							v'_2	...

FIGURE 1.1 – Comportement temporel du nœud sampling

formellement prouvée dans [For09]. La compilation repose notamment sur une série d'analyses statiques assurant la correction du programme.

Le *modèle de système* produit par PRELUDE consiste en un ensemble de tâches concurrentes et périodiques qui communiquent entre elles via des variables partagées stockées dans des buffers. Plus précisément, un système est défini comme un tuple $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$ où :

1. $\mathcal{S} = \{\tau_i\}_{i=1,\ldots,n}$ est un ensemble fini de tâches périodiques ;
2. $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{P} \times \mathcal{S}$ est la relation de précédence, où $\mathcal{P}$ est l'ensemble des *contraintes de précédence étendues périodiques* ;
3. $\mathcal{C} : \mathcal{S} \times \mathcal{V} \times \mathcal{S} \to \mathcal{W}$ est la fonction (partielle) de communication, où $\mathcal{W}$ est l'ensemble des *mots de dépendance* et $\mathcal{V}$ est l'ensemble des variables produites et consommées par les tâches. $\tau_i.in \subseteq \mathcal{V}$ (respectivement $\tau_i.out \subseteq \mathcal{V}$) correspond à l'ensemble des variables consommées (respectivement produites) par τ_i.

Tâches périodiques

Un ensemble de tâches périodiques $\mathcal{S} = \{\tau_i\}_{i=1,\ldots,n}$ est un ensemble fini de tâches, où chaque tâche est définie par les quatre attributs temps réel (O_i, T_i, D_i, C_i), vus dans la page d'introduction xiv et dont la signification sera expliquée dans le chapitre 2, page 12. Nous dénotons τ_i^k la $k^{ième}$ instance de τ_i (la première instance étant $k = 0$).

Contraintes de précédences étendues

Les tâches sont dépendantes car elles sont liées par la relation de précédence $\mathcal{R}$. Une *contrainte de précédence étendue* lie un ensemble d'instances des tâches communicantes. Si $\tau_i^n \to \tau_j^{n'}$ dénote une contrainte de précédence de l'instance n de τ_i à l'instance n' de τ_j, l'instance productrice doit nécessairement se terminer avant que la consommatrice ne débute. Pour n'importe quel $n \in \mathbb{N}$, $\mathcal{I}_n$ dénote l'ensemble d'entiers de l'intervalle $[0, n[$.

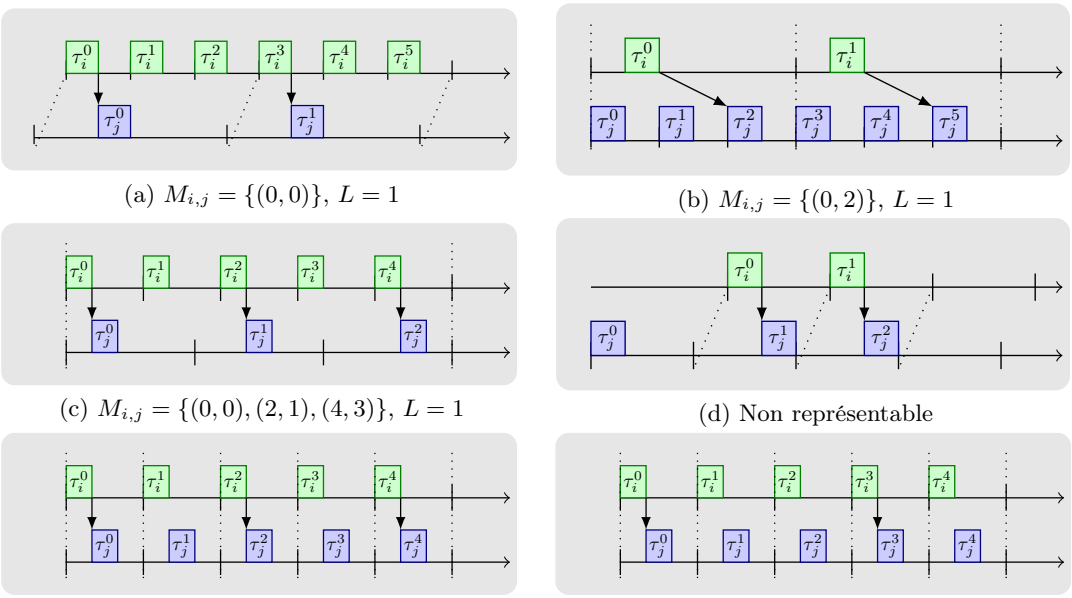
Définition 1 (Contraintes de précédences étendues périodiques)**.** *Si τ_i et τ_j sont deux tâches, $p = ppcm(T_i, T_j)$, $L \in \mathbb{N}^\star$ la longueur d'expression des précédences et $M_{i,j} \subseteq \mathcal{I}_{pL/T_i} \times \mathcal{I}_{pL/T_j}$. Une contrainte de précédence étendue périodique $\tau_i \xrightarrow{M_{i,j},L} \tau_j$ est définie comme l'ensemble des contraintes de précédence niveau instance $M'_{i,j}$:*

$$\forall (n, n') \in M'_{i,j}, \tau_i^n \to \tau_j^{n'}$$

avec $M'_{i,j} = \left\{ (n, n') \mid \exists k \in \mathbb{N}, (m, m') \in M_{i,j}, (n, n') = (m, m') + (k\frac{pL}{T_i}, k\frac{pL}{T_j}) \right\}$.

On parle de *contrainte de précédence simple* $\tau_i \rightarrow \tau_j$ si τ_i et τ_j ont la même période et si la contrainte de précédence périodique étendue est $M_{i,j} = \{(0,0)\}$ et $L = 1$.

Quelques exemples de contraintes de précédence sont représentés dans la figure 1.2. Les motifs sont exprimés sur le L fois le ppcm des périodes des deux tâches τ_i et τ_j. Dans 1.2(a) et 1.2(b), τ_i est 3 fois plus rapide que τ_j. Dans 1.2(a), l'instance 0 de τ_i s'exécute avant l'instance 0 de τ_j tous les ppcm des périodes des tâches. Sur le chronogramme 1.2(b), l'instance 0 de τ_i s'exécute avant l'instance 2 de τ_j. En 1.2(c), τ_i s'exécute 5 fois sur le ppcm et τ_j 3 fois. Sur le ppcm, il y a 3 précédences entre instance. En 1.2(d), τ_i et τ_j ont la même période mais le déphasage de τ_i est d'une période par rapport à τ_j. Cette contrainte n'est pas représentable avec la notion de contrainte de précédence étendue périodique car τ_j^0 se comporte différemment du reste des instances de la tâche. Dans les deux dernières figures 1.2(f), τ_i et τ_j ont la même période et les contraintes de précédence sont exprimées pour des longueurs $L > 1$. Ainsi, dans 1.2(e), deux tâches avec la même période communiquent avec la contrainte de précédence $M_{i,j} = \{(0,0)(0,1)\}$, c'est-à-dire le premier job de τ_i doit s'exécuter avant le premier et le deuxième job de τ_j. Cette contrainte est représentée pour $L = 2$. Dans 1.2(f), l'exemple est similaire sauf que la contrainte de précédence est exprimée en $L = 3$



(a) $M_{i,j} = \{(0,0)\}$, $L = 1$

(b) $M_{i,j} = \{(0,2)\}$, $L = 1$

(c) $M_{i,j} = \{(0,0),(2,1),(4,3)\}$, $L = 1$

(d) Non représentable

(e) $M_{i,j} = \{(0,0),(0,1)\}$, $L = 2$

(f) $M_{i,j} = \{(0,0),(0,1),(0,2)\}$, $L = 3$

FIGURE 1.2 – Précédences périodiques

Fonction de communication partielle Les communications entre 2 tâches sont totalement imposées par la sémantique du langage. Il y a une dépendance de donnée entre τ_i et τ_j s'il existe une variable $v \in \mathcal{V}$ telle que $v \in \tau_i.out$ et $v \in \tau_j.in$. Si deux tâches sont en dépendance de donnée, il y a nécessairement une précédence entre les deux tâches. Ainsi si $\tau_i^k \rightarrow \tau_j^n$, cela veut dire que la $n^{ième}$ instance de τ_j consomme la $k^{ième}$ valeur de v. Pour assurer que cette $k^{ième}$ valeur est toujours disponible, J. Forget propose de stocker les valeurs de v dans un buffer $b_{i,j,v}$ (idée inspirée de [STC06]) composé de plusieurs cases. Pour savoir où lire et où écrire, des mots de dépendance stockent l'information.

Le *mot périodique* $w = v(u)^\omega$ dénote la séquence infinie d'entiers composée par le préfixe v suivi par la répétition infinie de la séquence finie u. $w[k]$ dénote la $k^{ième}$ valeur de w.

Définition 2 (Mot de dépendance)**.** *Soient deux tâches τ_i, τ_j et une variable v tel que $v \in \tau_i.out$ et $v \in \tau_j.in$. Soit $b_{i,j,v}$ le buffer de communication entre τ_i et τ_j pour v. Les mots de dépendance $w = \mathcal{C}(\tau_i, v, \tau_j)$ et $w' = \mathcal{C}(\tau_j, v, \tau_i)$ sont les mots périodiques tel que $w[k]$ dénote l'indice de la case de $b_{i,j,v}$ où τ_i^k écrit et $w'[k]$ dénote la case où τ_j^k lit.*

Par exemple, si $\mathcal{C}(\tau_i, v, \tau_j) = 0(102)^\omega$, alors les valeurs produites par τ_i sont stockées dans le tampon comme suit : τ_i^0 n'est pas stocké, τ_i^1 est stocké dans la case 1, τ_i^2 n'est pas stocké, τ_i^3 est stocké dans la case 2, τ_i^4 est stocké dans la case 1, τ_i^5 n'est pas stocké (le motif 102 est répété indéfiniment).

Exemple 1. *Nous prenons comme exemple le nœud sampling (page 4). Le système généré par* PRELUDE *est $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$ avec :*

1. $\mathcal{S} = \{\tau_1 = (0, 10, 10, 2), \tau_2 = (0, 30, 30, 5), \tau_3 = (1, 60, 60, 30)\}$
2. $\mathcal{R} = \{(\tau_1, \{(0, 0)\}, \tau_2), (\tau_1, \{(0, 0)\}, \tau_3), (\tau_2, \{(0, 0)\}, \tau_3)\}$, $L = 1$ *pour toutes les contraintes,*
3. *Les variables échangées sont $\mathcal{V} = \{i, o_1, o_2, v_f, v_s\}$ tel que $\tau_1.in = \{i, v_s\}$, $\tau_1.out = \{o_1, v_f\}$, $\tau_2.in = \{v_f\}$, $\tau_2.out = \{v_s\}$, $\tau_3.in = \{v_s, v_f\}$ et $\tau_3.out = \{o_2\}$. Pour les communications, il faut un buffer à 1 case pour* `i` *($b_{-,1,i}$), un buffer à 2 cases $b_{1,2,vf}$ pour* `vf`*, un buffer à 2 cases $b_{2,1,vs}$ pour* `vs`*, un buffer à 2 cases $b_{1,3,vf}$ pour* `vf`*, un buffer à 2 cases $b_{2,3,vs}$ pour* `vs`*, un buffer à 1 case $b_{1,-,o1}$ pour* `o1` *et un buffer à 1 case $b_{3,-,o2}$ pour* `o2`*. Les lectures/écritures sont définies dans le tableau ci-dessous :*

	i	o_1	o_2	v_f	v_s
τ_1	(1)	(1)		$b_{1,2,v_f}$: (100) $b_{1,3,v_f}$: (100000200000)	$b_{2,1,v_s}$: (111222)
τ_2				$b_{1,2,v_f}$: (1)	$b_{2,1,v_s}$: (21) $b_{2,3,v_s}$: (1020)
τ_3			(1)	$b_{1,3,v_f}$: (12)	$b_{2,3,v_s}$: (12)

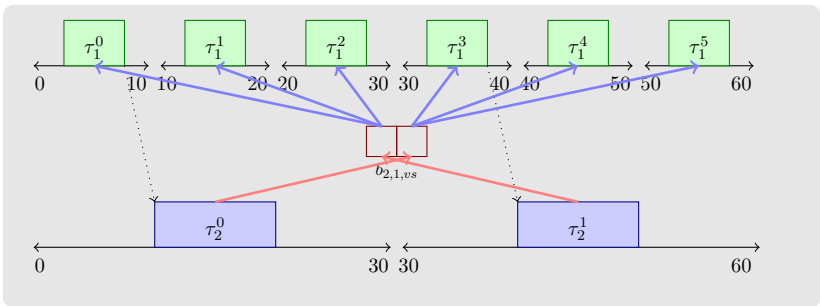


FIGURE 1.3 – Communication de `vs` entre τ_2 et τ_1

Nous expliquons les mots pour v_s. La figure 1.3 montre la communication de `vs` *entre τ_2 et τ_1. On montre les jobs (6 pour τ_1, 2 pour τ_2), leur intervalle d'exécution, les précédences entre eux par des flèches en pointillé, les buffers de communication, les lectures par des flèches bleues et les écritures par des flèches rouges. v_s est stocké dans deux tampons différents $b_{2,1,v_s}$ pour $\tau_2 \to \tau_1$ et $b_{2,3,v_s}$ pour $\tau_2 \to \tau_3$. v_s est produite une fois par τ_2 pendant que τ_1 s'exécute trois fois. v_s est consommé en décalé grâce à un* `fby` *, pourtant le tampon est composé de deux cases et la valeur initiale est dans la première case. Pendant son exécution, τ_2^0 stocke la valeur dans la deuxième case ce qui n'interfère pas avec l'exécution en cours de τ_1. τ_2^1 écrit dans la case 1 pendant qu'à son tour τ_1 lit trois fois la valeur précédente dans la case 2. Ce comportement se répète indéfiniment.*

La figure 1.4 montre la communication de `vs` *entre τ_2 et τ_3. Le décalage de phase impose d'avoir un buffer à deux cases car τ_2^1 peut écrire la nouvelle valeur de* `vs` *juste avant que τ_3^0 ne lise l'ancienne (ce comportement peut se produire pendant l'intervalle $[60, 61]$).*

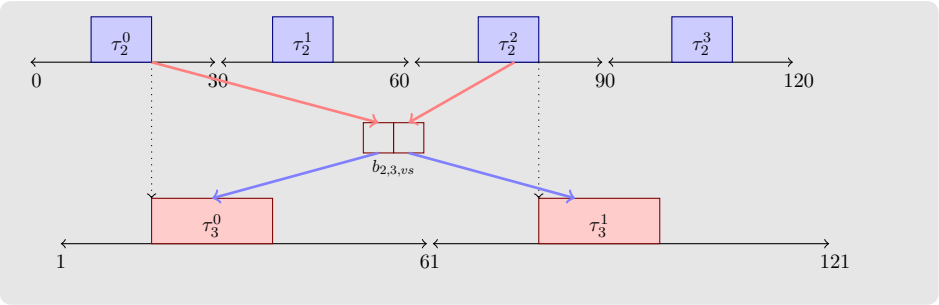


FIGURE 1.4 – Communication de `vs` entre τ_2 et τ_3

1.3 Exécution monoprocesseur

Un compilateur de PRELUDE a été développé par Julien Forget en *OCaml* et est disponible sur son site web [1]. Le compilateur génère un ensemble de tâches et le traduit en code C pour une exécution monoprocesseur. Le code généré est un ensemble de fonctions pouvant chacune être exécutées au sein d'un thread POSIX. Les tâches sont codées comme des threads communicants qui s'exécutent indéfiniment en boucle. Chaque tâche utilise les buffers nécessaires pour ses communications. L'ensemble de tâches obtenu peut être exécuté sur MARTE OS [RH01]. Pour cela, un ordonnanceur RM [FBLP10] et un autre EDF sont disponibles avec le protocole de communication.

Exemple 2. *Continuons à traiter l'exemple* sampling *de la page 4.*

```
int tau_1_o2_tau_3_i0[2];
int tau_2_o1_tau_3_i1[2];
int tau_3_o1_o2;
```

Les buffers de communication sont codés par des variables globales : `tau_1_o2_tau_3_i0` *correspond au buffer* $b_{1,3,vf}$ *pour échanger la variable* `vf` *entre* τ_1 *et* τ_3*,* `tau_2_o1_tau_3_i1` *à* $b_{2,3,vs}$ *et* `tau_3_o1_o2` *stocke la variable de sortie* `o2`*.*

La fonction d'une tâche contient le protocole de communication et permet d'encapsuler la fonction du nœud importé. Chaque exécution correspond à un job de la tâche. Le code généré pour le nœud `tau_3` *est :*

```
void tau_3_fun(void* args)
{
  static int i0_rcell=0;
  static int i1_rcell=0;

  tau_3_o1_o2=tau_3(tau_1_o2_tau_3_i0[i0_rcell],tau_2_o1_tau_3_i1[↩
      i1_rcell]);
  i0_rcell=(i0_rcell+1)%2;
  i1_rcell=(i1_rcell+1)%2;
}
```

La fonction d'une tâche calcule la sortie `o2`*, et manipule deux variables* `i0_rcell` *et* `i1_rcell` *pour indiquer la case des buffers à lire.*

1. Le compilateur PRELUDE est disponible sur le site http://www.lifl.fr/{~{}}forget/prelude.html

La fonction `main` *gère la création de tous les threads d'exécution. Chaque tâche est associée à un thread, en passant le pointeur de sa fonction en paramètre et son comportement temporel est déterminé par des attributs temps réel spécifiques.*

1.4 Résumé

PRELUDE est un langage formel pour la conception de systèmes de contrôle-commande qui permet de décrire l'architecture logicielle d'un système avec une sémantique synchrone. Une fois l'architecture complète compilée, on obtient un ensemble de tâches concurrentes. Ces tâches sont caractérisées par un ensemble de paramètres temps réel ainsi que des contraintes de précédence. Une exécution sur une architecture monoprocesseur est proposée, en utilisant soit un ordonnanceur RM soit un ordonnanceur EDF sur le système d'exploitation temps réel MARTE OS.

Chapitre 2

Ordonnancement multiprocesseur

La première question qui se pose une fois l'ensemble de tâches généré est de savoir s'il est possible de l'exécuter correctement sur une cible multiprocesseur. Il faut donc pouvoir affecter les ressources processeur à chaque tâche et à chaque instant de façon à respecter toutes les contraintes temporelles et de précédence. Le domaine qui s'intéresse à cette problématique s'appelle l'*ordonnancement temps réel* et a été largement étudié depuis les années 60. L'ordonnancement *monoprocesseur* a occupé une grande part des recherches, mais l'émergence des *multicœurs* et des *manycœurs* a favorisé les dernières recherches en multiprocesseur. Nous présentons dans ce chapitre les résultats importants qui répondent à notre problème. A noter que l'ordonnancement ne gère que $(\mathcal{S}, \mathcal{R})$ de l'ensemble de tâches généré et suppose que les communications $\mathcal{C}$ sont réalisées en mémoire partagée selon un protocole indépendant de l'ordonnancement.

2.1 Modèle de la cible et des tâches

Architecture cible Depuis 5-6 ans, les architectures multicœur (plusieurs processeurs intégrés sur une même puce et communicant par un bus interne) ont envahi le marché. Ces architectures sont apparues du fait d'un besoin croissant en puissance et de la difficulté technique à produire des monoprocesseurs encore plus rapides [BC11]. Les architectures multicœur peuvent être classifiées en trois catégories :

- *Hétérogènes :* Les processeurs sont différents. Par conséquent, le rythme d'exécution d'une tâche dépend du processeur sur lequel elle s'exécute ;
- *Uniformes :* Les processeurs s'exécutent à des vitesses proportionnelles. Pour deux processeurs, p_1, p_2 tels que le rapport de vitesse est $v_1 = 2v_2$, le rythme d'exécution d'une tâche est toujours le double dans p_1 ;
- *Homogènes :* Les processeurs sont identiques. Le rythme d'exécution des tâches est le même sur tous les processeurs.

La plupart des puces multicœur commercialisées appartiennent aux architectures homogènes, sur lesquelles nous nous concentrons dans la suite. Les architectures considérées sont alors composées de m processeurs identiques $\mathcal{P} = \{p_1, p_2, ..., p_m\}$.

Dans la littérature on trouve les termes multiprocesseur et multicœur. Une architecture multiprocesseur est composée de plusieurs processeurs reliés par un medium de communication. Une architecture distribuée peut être vue comme un multiprocesseur. Un multicœur est également un cas particulier d'un multiprocesseur. En théorie de l'ordonnancement, on parle généralement de multiprocesseur et à l'exécution on précise la cible (donc multicœur dans notre contexte). A noter également qu'en théorie de l'ordonnancement, on fait souvent l'hypothèse que le coût d'un

changement de contexte (arrêt d'une tâche, chargement d'une tâche en mémoire,...) et le coût d'une migration ou d'une communication en mémoire partagée sont nuls.

Modèle de tâches Les systèmes de type contrôle-commande manipulent généralement des ensembles de tâches *périodiques*, c'est-à-dire chaque tâche est relancée à une période de temps fixe. Le modèle de tâches considéré dans ce document est celui déjà évoqué dans l'introduction. Un système $\mathcal{S}$ est composé de n tâches : $\mathcal{S} = \{\tau_i\}_{i=1,\ldots,n}$. Chaque tâche est contrainte par quatre paramètres temporels, illustrés dans la figure 2.1 :

- O_i : date de réveil de la tâche. Lorsque toutes les tâches commencent à l'instant initial, $\forall i = 0..n, O_i = 0$ on parle de système *synchrone*. A l'inverse, si au moins une tâche commence plus tard, on parle de système *asynchrone*.
- T_i : la période de répétition. Une tâche se répète infiniment dans le temps avec cette période de répétition. Chaque nouvelle activation d'une tâche s'appelle *instance* ou *job*. Un nouveau *job* est démarré à chaque nouvelle période. On note, dans la suite, τ_i^k pour la $k^{ième}$ instance de la tâche i ;
- D_i : l'échéance d'exécution. La tâche doit terminer son exécution avant cette échéance. Lorsque $\forall i = 0..n, T_i = D_i$ on parle d'un ensemble à *échéances implicites*, lorsque $\exists i = 0..n, T_i \geq D_i$ on parle d'ensemble à *échéances contraintes*. Lorsque il n'y a pas de relation entre les périodes et les échéances (les échéances peuvent être supérieures, inférieures ou identiques aux périodes) on parle alors d'*échéances arbitraires*. Dans cette thèse on ne considère que des ensembles de tâches à échéances contraintes ou implicites ;
- C_i : l'estimation du pire temps d'exécution, ou WCET (Worst Case Execution Time). Ces temps sont difficiles à calculer et peuvent parfois être très pessimistes. Pour en savoir plus sur le calcul des pires temps d'exécution plusieurs publications sont disponibles [UCS+10, WEE+08]. Dans le cadre de notre travail le WCET est une donnée du problème d'ordonnancement.

On peut, donc, définir une tâche par $\tau = (O, T, D, C)$. Ce modèle est une extension du modèle classique de Liu et Layland [LL73] où une tâche n'était alors définie que par sa période et son pire temps d'exécution. On doit remarquer que ce modèle est un cas particulier du modèle *sporadique*, où chaque tâche doit se relancer, au plus tôt, à une période de temps choisie. L'*hyper-période* H est le plus petit commun multiple des périodes de toutes les tâches, ($H = ppcm(T_i)$).

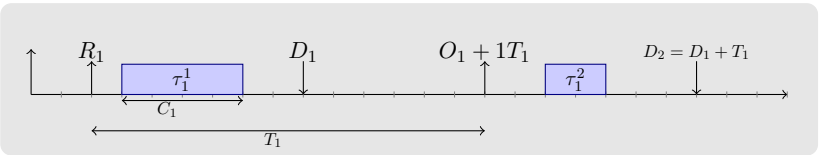


FIGURE 2.1 – Modèle de tâche

Le facteur d'utilisation u_i d'une tâche est C_i/T_i et représente le pourcentage de processeur nécessaire pour l'exécuter. Sachant que la capacité d'un processeur est de 1, une architecture à m processeurs aura une capacité "théorique" de m. La charge totale du système est la somme de la charge de toutes les tâches, $U = \sum_{\tau_i \in \mathcal{S}}(u_i)$. Le *pire temps de réponse* d'une tâche est défini comme le temps le plus long entre l'activation d'un job et la fin de l'exécution de ce job.

On parle d'ensemble de tâches *dépendantes* si les tâches sont soumises à des contraintes de précédence. Sinon on parle de tâches *indépendantes*. Comme expliqué dans le chapitre 1, les tâches dans notre contexte sont soumises à des contraintes de précédence généralisées.

2.2 Politiques d'ordonnancement

Afin d'exécuter les tâches sur les m processeurs en respectant les contraintes temporelles et de précédence, on utilise des politiques (ou algorithmes, ou stratégies) d'ordonnancement. Ces dernières sont responsables de décider à tout instant t quelle(s) tâche(s) doit(vent) être exécutée(s) sur les processeurs.

En théorie d'ordonnancement il y a deux conditions de base : un processeur ne peut exécuter qu'une seule tâche à un instant t et une tâche ne peut s'exécuter que sur un seul processeur à un instant t (pas de parallélisme intratâche). Les politiques dites "gang scheduling" [FR92] permettant du parallélisme intra-tâche sont hors du sujet de ce travail.

2.2.1 Définitions

Plusieurs concepts permettent de caractériser les politiques d'ordonnancement.

Hors-ligne/En-ligne

On peut classifier les politiques d'ordonnancement en deux groupes : les politiques *en-ligne* et celles *hors-ligne*. Les premières décident à l'exécution les tâches à exécuter et sur quel processeur. Les deuxièmes calculent l'ordonnancement avant l'exécution qui se réalisera ensuite par un séquenceur.

Préemption

Une politique d'ordonnancement est *préemptive*, si une tâche en exécution peut être interrompue avant de finir son exécution et reprendre plus tard. Si l'interruption n'est pas permise on parle de politiques d'ordonnancement *non préemptives*.

Migration

Sur une architecture multiprocesseur, une tâche peut commencer son exécution sur un processeur et poursuivre sur un autre processeur. On dit alors qu'il y a migration. On peut distinguer trois types de migration illustrés dans la figure 2.2 :

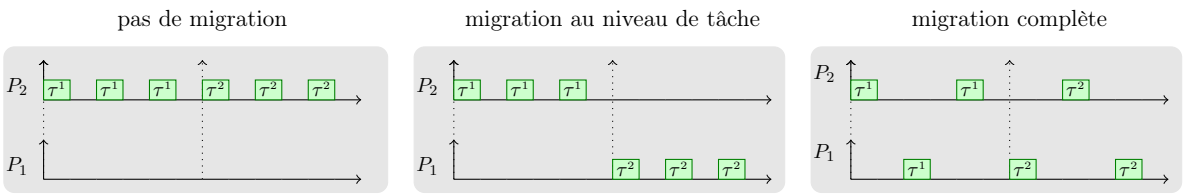
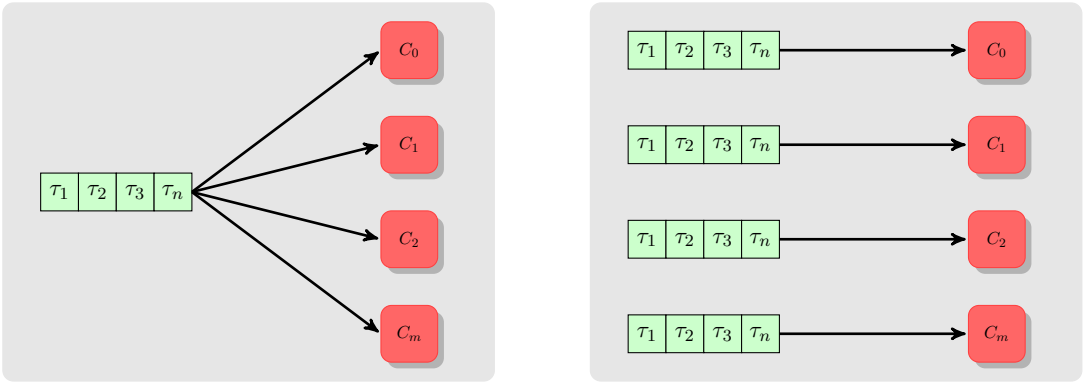


FIGURE 2.2 – Illustration des 3 niveaux de migration

- *migration non permise* : une tâche est placée sur un processeur et doit toujours s'exécuter sur ce processeur ;
- *migration permise au niveau tâche* : l'instance d'une tâche doit s'exécuter pendant toute son activation sur le même processeur. Cependant, chaque instance peut s'exécuter sur des processeurs différents ;
- *migration permise (au niveau instance ou migration complète)* : toute tâche interrompue peut reprendre son exécution sur n'importe quel processeur.

Les politiques d'ordonnancement qui ne permettent pas la migration sont aussi appelées dans la littérature politiques *partitionnées*. L'ordonnancement partitionné se fait en général en deux étapes : (a) division de l'ensemble de tâches en plusieurs sous-ensembles à l'aide d'une heuristique [LGDG00] et allocation de chaque sous-ensemble à un processeur et (b) ordonnancement monoprocesseur sur chaque cœur. La création de sous-ensembles de tâches est analogue au problème de *bin packing* [GGJY76]. Ce problème classique a été bien étudié dans la littérature et est NP-complet.



(a) Exécution globale (b) Exécution partitionnée

FIGURE 2.3 – Exécution globale vs. exécution partitionnée

Les politiques à migration complète sont dites *globales*. En effet, cela revient à dire qu'il y a une unique file pour stocker les tâches actives en attente et que les décisions sont prises en tenant compte "globalement" des tâches. Sur la figure 2.3, les modèles d'exécution global et partitionné sont représentés.

Conservatif

Un algorithme d'ordonnancement *conservatif* ne laisse pas de processeurs oisifs s'il existe une ou plusieurs tâches actives requérant un processeur.

Choix des priorités

Les politiques d'ordonnancement en-ligne se servent classiquement de *priorités* : elles affectent une priorité à chaque tâche et exécutent les m (soit le nombre de processeurs) tâches les plus prioritaires si le nombre de tâches actives est supérieur au nombre de processeurs. Sinon, on exécute toutes les tâches actives. L'affectation de priorités peut se produire de trois façons différentes illustrées dans la figure 2.4 :

- *priorité fixe :* les priorités sont affectées aux tâches (donc, toutes les instances d'une tâche ont la même priorité) au début de l'exécution et sont conservées au long de l'exécution ;
- *priorité fixe au niveau de l'instance :* la priorité ne peut pas changer pendant toute l'activation d'une instance mais entre deux instances différentes la priorité peut changer ;
- *priorité dynamique :* les priorités de toutes les tâches peuvent changer à tout instant.

On peut diviser le problème de l'ordonnancement multiprocesseur en deux sous-problèmes : l'*affectation de priorités* et l'*allocation sur les processeurs*. Carpenter [CFH+04] a proposé un tableau récapitulatif des familles de politiques d'ordonnancement préemptives.

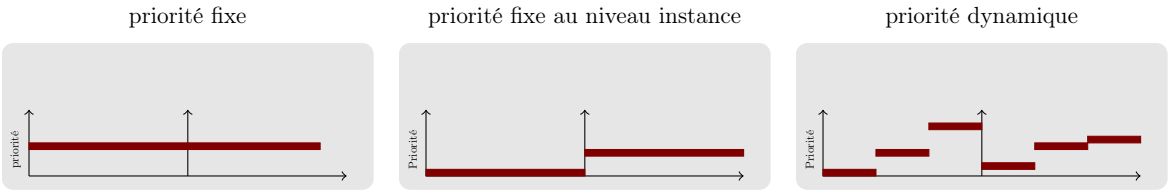


FIGURE 2.4 – Illustration des 3 niveaux de priorité

	1 : Priorité fixe	2 : Priorité fixe niveau instance	3 : Priorité dynamique
3 : migration permise	(1,3)	(2,3)	(3,3)
2 : migration niveau de tâche	(1,2)	(2,2)	(3,2)
1 : pas de migration	(1,1)	(2,1)	(3,1)

FIGURE 2.5 – Classification de Carpenter

2.2.2 Politiques existantes pour architectures multiprocesseur

Dans ce travail de thèse, nous n'avons considéré que des politiques d'ordonnancement globales et préemptives. On s'intéresse donc à la première ligne de tableau du 2.5. Deux types de politiques globales multiprocesseur existent : les politiques classiques (celles définies pour l'ordonnancement monoprocesseur) adaptées et les politiques spécifiques aux architectures multiprocesseur. Les politiques classiques peuvent être aisément étendues pour les architectures multiprocesseur. Lorsque on étend une politique d'ordonnancement monoprocesseur, on ajoute la lettre "g" avant le nom (pour indiquer global).

Nous nous intéressons seulement à quatre politiques en-ligne : une représentative des politiques à priorité fixe, une deuxième représentative des politiques à instances à priorité fixe et deux représentatives de politiques à priorité dynamique. Pour en savoir plus sur d'autres politiques, plusieurs résultats ont été développés et résumés dans les études [SSNB94, SAA+04, BB07, DB09].

FP (Fixed Priority) est la politique la plus simple et la préférée des industriels. La priorité est choisie hors-ligne et est unique à chaque tâche. Cette politique est de type (1,3) dans le tableau 2.5. Dans ce groupe de politiques à priorité fixe, deux ont été plus particulièrement étudiées : RM (Rate Monotonic) présentée par Liu et Layland [LL73] affectant la priorité la plus grande à la tâche avec la période la plus petite et DM (Deadline Monotonic) introduite par Leung et Whitehead [LW82] affectant la priorité la plus élevée à la tâche avec l'échéance la plus petite.

Exemple 3. *On considère l'ensemble de tâches suivant :*

τ_i	O_i	T_i	D_i	C_i
τ_0	0	4	4	2
τ_1	0	8	8	6
τ_2	0	16	16	8

L'exécution décrite sur le chronogramme 2.6 montre l'exécution sur une architecture à deux processeurs. La priorité d'exécution est affectée selon l'ordre dans le tableau, c'est-à-dire, $\tau_0 < \tau_1 < \tau_2$. Les premières tâches à commencer leur exécution sont les deux les plus prioritaires, τ_0, τ_1. La tâche τ_2, avec la priorité la plus basse, s'exécute dans les créneaux laissés par τ_0 sur le premier processeur en étant préemptée à chaque fois que la tâche τ_0 initialise une nouvelle instance. Le deuxième et le quatrième démarrage de τ_2^0 pourraient migrer et s'exécuter sur l'autre processeur.

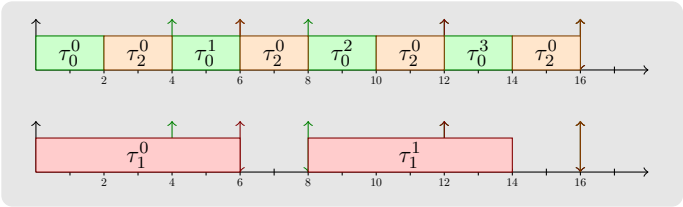


FIGURE 2.6 – Exécution en FP

gEDF (Earliest Deadline First) a été également introduit par Liu et Layland [LL73]. Ici, la priorité est donnée en fonction de la prochaine échéance absolue définie comme $d_i^k = O_i + kT_i + D_i$. La priorité la plus grande est pour la tâche avec l'échéance la plus proche. Les priorités ne changent pas tout au long de l'exécution des instances par conséquent, on se trouve dans le cas (2,3) du tableau 2.5.

Exemple 4. *On reprend le modèle de tâches utilisé pour illustrer l'exemple 3. La figure 2.7 correspond à l'exécution avec un ordonnanceur gEDF sur 2 processeurs. Étant donné que les échéances de τ_0 et τ_1 sont plus petites que celle de τ_2, jusqu'à 12 les deux premières tâches sont prioritaires, en conséquence dans cet intervalle la séquence d'exécution est similaire à FP. En revanche, à 12 l'échéance de τ_2^0, $d_2^0 = 16$, est similaire à celle de τ_0^3, $d_0^3 = 16$ et τ_1^1, $d_1^1 = 16$, alors, les trois tâches peuvent être exécutées. Afin de réduire les nombre de préemptions, on choisit d'exécuter les tâches qui sont déjà en exécution, c'est-à-dire τ_1^1 et τ_2^0 pour reprendre à la fin de τ_2^0 le job τ_0^3.*

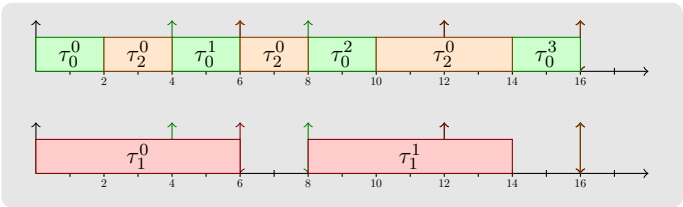


FIGURE 2.7 – Exécution en gEDF

gLLF (Least Laxity First) a été introduit par Mok [Mok83]. La priorité la plus grande est affectée à la tâche avec la laxité la plus petite, soit le temps d'exécution restant jusqu'à la prochaine échéance absolue à un instant t. Plus formellement, $l_\tau(t) = D_\tau - C_\tau(t) - ((t - O_\tau) \ mod \ T_\tau)$ où $C_\tau(t)$ est le temps d'exécution restant à l'instant t jusqu'à la prochaine échéance. La priorité peut changer à chaque instant et donc cette politique appartient à la famille totalement dynamique (3,3) du tableau 2.5.

Exemple 5. *L'ensemble de tâches considéré est à nouveau celui de l'exemple 3. On montre le calcul des laxités dans le tableau suivant :*

instant	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
l_{τ_0}	2	2	-	-	2	2	-	-	2	2	-	-	2	1	1	-
l_{τ_1}	2	2	2	2	2	2	-	-	2	2	2	2	2	2	-	-
l_{τ_2}	8	7	6	6	6	5	4	4	4	3	2	2	2	2	1	-

et l'exécution sur deux processeurs dans le diagramme de Gantt :

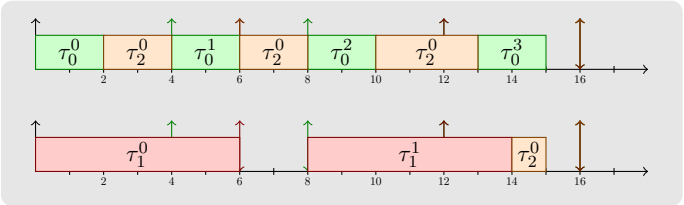


FIGURE 2.8 – Exécution en gLLF

L'exécution commence avec l'affectation de τ_0^0 et τ_1^0 sur les processeurs car elles ont les laxités les plus petites. A l'instant 2 l'instance τ_0^0 termine son exécution et la tâche τ_2^0 peut occuper un des processeurs. A l'instant 4 l'instance τ_0^1 préempte τ_2^0 et commence son exécution en parallèle de τ_1^0. A 6, τ_0^1 termine son exécution et laisse sa place à τ_2^0. A 8, τ_0^2 et τ_1^1 commencent un nouveau job et ont une laxité plus petite que τ_2^0. A 10, τ_2^0 prend la place de τ_0^2 qui se termine. A 12 les laxités de toutes les tâches sont identiques alors, les tâches τ_2^0 et τ_1^1 continuent. A 13, la tâche τ_0^3 a une laxité plus petite et préempte τ_2^0. A 14 la tâche τ_1^1 a fini son exécution et τ_2^0 migre pour s'exécuter sur le processeur libéré.

LLREF (Largest Local Remaining Execution First) a été introduit par Cho et al. [CRJ06] spécifiquement pour les architectures multiprocesseur. Cette politique mélange les idées de Baruah and al. de PFair [Bar95] et celles de laxité de LLF. En PFair, les tâches progressent à un rythme constant. En LLREF, l'exécution est divisée en quanta fixes de temps dont la taille dépend de la prochaine date de réveil ou de la prochaine échéance d'une instance. Soit T' la prochaine date, égale à la prochaine soit date de réveil soit échéance : $\forall i = 0..n \ T' = min(t_i, o_i)$ où t_i est la prochaine période et o_i la prochaine date de réveil en temps absolu de τ_i. Deux paramètres sont utilisés pour affecter la priorité aux différentes tâches :

1. l_τ représente la *local remaining execution time* de τ dans l'intervalle $[t, T']$. Correspond au taux d'exécution équitable qui doit être réalisé :

$$l_\tau(t) = C_\tau(t) \times \frac{T' - t}{T_\tau - t} \tag{2.1}$$

2. L_τ est la *laxité locale* de τ dans l'intervalle $[t, T']$:

$$L_\tau(t) = T' - t - l_\tau(t) \tag{2.2}$$

Cette politique fonctionne comme suit :

1. lorsque la laxité est nulle, $L_\tau(t) = 0$, il faut utiliser le temps restant pour compléter la tâche τ. Ainsi, l'exécution est urgente et on affecte la priorité la plus élevée ;
2. pour l'ensemble des tâches avec une laxité locale strictement positive, la priorité la plus élevée est affectée aux tâches avec le taux d'exécution équitable l_τ le plus grand.

Si une tâche est en exécution à t elle peut occuper un processeur jusqu'en t' lorsqu'une tâche (a) termine son temps d'exécution, (b) devient urgente ou (c) se réveille.

$$t' = t + \min(\max_{i=0..n}(L_i), \min_{i=0..n}(l_i)) \tag{2.3}$$

Exemple 6. *Nous illustrons l'algorithme LLREF avec l'ensemble de tâches de l'exemple 3 et sur deux processeurs. A l'instant 0, l'échéance la plus petite est $T' = 4$. On peut calculer les paramètres des tâches. Par exemple, pour τ_1 on a :*

$$l_{\tau 0}(0) = 2 \times \frac{4-0}{4-0} = 2$$

$$L_{\tau 0}(0) = 4 - 0 - 2 = 2$$

Le diagramme de Gantt est donné dans la figure 2.9 et les valeurs des paramètres sont données ci-dessous. En 0, aucune laxité locale n'est nulle, donc aucune tâche n'est urgente. On se concentre sur l_τ pour affecter les priorités des tâches. Les deux jobs avec le temps restant d'exécution local le plus grand sont τ_0^0 et τ_1^0. Elles occupent donc les processeurs jusqu'à 2 où τ_0^0 termine son exécution. A cette date, il reste 2 tâches actives qui occupent les processeurs. A 4 la tâche τ_0^1 commence une autre instance mais comme les l_τ sont identiques, elle ne peut s'exécuter qu'entre 6 et 8. A partir de 8 la nouvelle instance τ_1^1 prend un processeur et l'autre est partagé entre les deux instances de τ_0 qui s'activent et les deux unités de temps restant de τ_2^1. t' n'est calculé que lorsqu'il y a plus de tâches actives que de processeurs, auquel cas la prochaine date de calcul des paramètres est T'.

t	0	2	4	6	8	10	12	14
$T'(t)$	4	4	8	8	12	12	16	16
$l_{\tau 0}(t)$	2	–	2	2	2	–	2	–
$L_{\tau 0}(t)$	2	–	2	0	2	–	2	–
$l_{\tau 1}(t)$	3	4/3	2	–	3	4/3	2	–
$L_{\tau 1}(t)$	1	2/3	2	–	1	2/3	2	–
$l_{\tau 2}(t)$	2	8/7	2	4/5	1	2/3	–	–
$L_{\tau 2}(t)$	2	6/7	2	6/5	3	4/3	–	–
t'	2	-	6	-	10	-	14	-

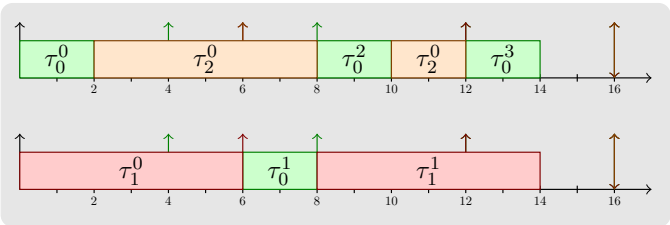


FIGURE 2.9 – Exécution en LLREF

2.2.3 Précédences

Pour ordonnancer des tâches dépendantes, on utilise généralement des sémaphores. Ces mécanismes engendrent un certain nombre de problèmes comme l'inversion de priorité qu'il faut traiter.

Pour éviter ces problèmes, Chetto et al. dans [CSB90] s'intéressent à coder les précédences simples d'un ensemble de jobs pour un ordonnanceur EDF en monoprocesseur en jouant sur les attributs des jobs. A partir des études de Blazewicz [Bla77], les auteurs trouvent une solution optimale pour garantir les contraintes d'exécution des jobs temps réel avec précédences.

Soit un ensemble de jobs $\mathcal{S} = \{\tau_i = (O_i, D_i, C_i)\}_{i=1,\dots n}$ soumis à des contraintes de précédence simples $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$, l'algorithme de Chetto et al. consiste à modifier les échéances de manière à forcer l'ordre d'exécution de certains jobs, même si ces derniers ne sont pas les plus prioritaires, comme suit :

$$D_i^* = min(D_i; \min_{\tau_j \in succ(\tau_i)} (D_j^* - C_j))$$

où *succ* représente les tâches qui succèdent τ_i suivant la relation de précédence. On modifie également les dates de réveil selon la formule :

$$O_i^* = max(O_i; \max_{\tau_j \in prec(\tau_i)} (O_j^* + C))$$

où *prec* représente les tâches qui précèdent τ_i suivant la relation de précédence simple.

Exemple 7. *Considérons un ensemble de tâches soumises à précédences décrit dans le tableau suivant :*

τ_i	O_i	D_i	C_i	O_i^*	D_i^*
τ_1	0	10	3	0	4
τ_2	0	10	4	3	8
τ_3	0	10	1	0	3
τ_4	0	10	2	7	10
τ_5	0	3	3	0	3

$\mathcal{R}$

$\tau_1 \rightarrow \tau_2$

$\tau_2 \rightarrow \tau_4$

Les résultats de la méthode de Chetto et al. se trouvent dans la partie droite du tableau. Les nouvelles valeurs sont obtenues en suivant les formules :

$$\begin{aligned}
O_1^* &= max(O_1) = 0 \\
O_2^* &= max(O_2; O_1^* + C_1) = max(0; 3) = 3 \\
O_4^* &= max(O_4; O_2^* + C_2) = max(0; 7) = 7 \\
D_4^* &= min(D_4) = 10 \\
D_2^* &= min(D_2; D_4^* - C_4)) = min(10; 8) = 8 \\
D_1^* &= min(D_1; D_2^* - C_2) = min(10; 4) = 4
\end{aligned}$$

En monoprocesseur, il suffit que O^* soit plus grand que la date de réveil des prédécesseurs.

Ces travaux ont été étendus dans [FBG+10] dans le cas d'un ensemble de tâches multipériodiques soumises à des contraintes de précédence généralisées et pour des ordonnancements à priorité fixe. Les auteurs ont proposé d'appliquer DM avec modification des paramètres dans le cas des ensembles de tâches synchrones et Audsley [Aud91] modifié pour des ensembles de tâches asynchrones. Il faut légèrement modifier les formules pour prendre en compte la périodicité : $D_i^* = D_i + O_i - O_i^*$. Le codage des précédences dans les paramètres est sous-optimal : en effet, il existe des ensembles de tâches ordonnançables avec sémaphores et priorité fixe qui ne le sont

pas par encodage. Dans [FGPR11], les auteurs ont proposé des stratégies optimales monoprocesseur EDF dans le cas d'un ensemble de tâches multipériodiques soumises à des contraintes de précédence généralisées par modification des paramètres (à la Chetto et al.).

Malheureusement, l'application de ces techniques en multiprocesseur n'est pas directe.

Exemple 8. *Pour l'ensemble de tâches donné dans l'exemple 7 et une politique d'ordonnancement gEDF on a une exécution erronée en multiprocesseur en prenant l'encodage de Chetto. Le job τ_2 commence à l'instant 3 avant la fin de l'exécution de τ_1 comme le montre la figure 2.10.*

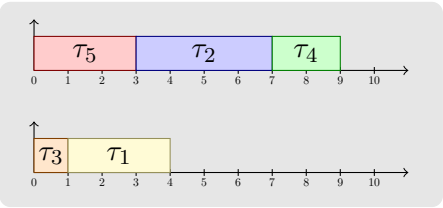


FIGURE 2.10 – Encodage avec Chetto en multiprocesseur

Le problème vient évidemment du parallélisme d'exécution par rapport à la séquence en monoprocesseur. La modification des échéances est correcte mais pour les dates de réveil, il faut en réalité forcer le démarrage de la tâche successeur après le pire temps de réponse de la tâche prédécesseur.

2.2.4 Optimalité

Une autre notion particulièrement intéressante en théorie de l'ordonnancement est celle de l'optimalité. L'idée est d'évaluer une politique ou un algorithme selon sa capacité à ordonnancer correctement le plus d'ensembles de tâches. On rappelle plusieurs définitions liées à la notion d'optimalité.

Définition 3 (Faisabilité)**.** *Un ensemble de tâches est* faisable *s'il existe un algorithme d'ordonnancement qui est capable d'ordonner les tâches correctement, c'est-à-dire en respectant les contraintes temporelles et les contraintes de précédence.*

Définition 4 (Optimalité)**.** *Une politique d'ordonnancement est* optimale *si la politique ordonnance tous les ensembles de tâches faisables. On dit qu'une politique est optimale pour une famille de modèles de tâches et une famille de politiques (cf tableau 2.5) si elle est capable d'ordonnancer tout ensemble de tâches appartenant à la famille faisable par la famille de la politique.*

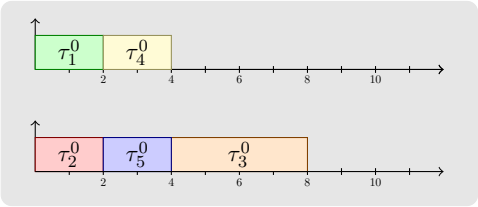
Définition 5 (Clairvoyance)**.** *Une politique d'ordonnancement est dite* clairvoyante *si, pour décider l'affectation de priorités, elle utilise les événement futurs. Dans notre contexte, les événements futurs sont les prochaines échéances et les prochaines dates de réveil. Les algorithmes en-ligne, n'ayant connaissance que des tâches en cours d'exécution, ne sont pas clairvoyants. Pour être optimale sur une architecture multiprocesseur et pour des tâches asynchrones ou avec des échéances contraintes une politique doit être clairvoyante [HL88].*

Exemple 9. *Prenons un exemple publié dans [HL88]. Soit l'ensemble de tâches donné dans la figure 2.11(a) pour qu'aucune tâche ne rate ses contraintes temporelles, les tâches τ_1 et τ_2 doivent prendre les priorités les plus élevées à l'instant 0. Par contre, comme illustré dans 2.12, si on*

change les paramètres de τ_4 et τ_5 en les faisant commencer plus tard, pour respecter toutes les contraintes temporelles, τ_1 et τ_3 doivent prendre les priorités les plus élevées à l'instant 0*. Par conséquent, deux tâches qui démarrent leur exécution plus tard peuvent modifier la décision à prendre pour un ordonnancement courant, autrement dit, il faut être* clairvoyant*.*

τ_i	O_i	T_i	D_i	C_i
τ_1	0	20	4	2
τ_2	0	20	4	2
τ_3	0	20	8	4
τ_4	2	20	2	2
τ_5	2	20	2	2

(a)

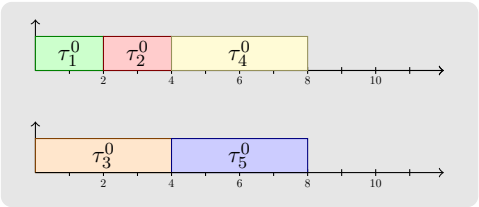


(b)

FIGURE 2.11 – Exemple Hong

τ_i	O_i	T_i	D_i	C_i
τ_1	0	20	4	2
τ_2	0	20	4	2
τ_3	0	20	8	4
τ_4	4	20	4	4
τ_5	4	20	4	4

(a)



(b)

FIGURE 2.12 – Exemple Hong

Quelques résultats d'optimalité

Plusieurs travaux offrent des résultats par rapport à l'optimalité des politiques d'ordonnancement existantes. Pour les politiques à priorité fixe, [LL73] montre que RM est optimal pour un ensemble de tâches indépendantes synchrones à échéance implicite sur une architecture monoprocesseur. Dans [LW82], l'auteur montre que DM est optimal pour des ensembles de tâches indépendantes synchrones à échéance contrainte et synchrone sur une architecture monoprocesseur. En revanche, aucune de ces politiques ne reste optimale sur des architectures multiprocesseur [DB09].

Dans le cas de politiques à priorités dynamiques, EDF [LL73] et LLF [LW82, Mok83] sont optimales pour des ensembles de tâches indépendantes asynchrones à échéance contrainte sur des architectures monoprocesseur mais, comme dans le cas des priorités fixes, ces résultats ne sont pas transposables en multiprocesseur. Les politiques PFair [BCPV94] et LLREF [CRJ06] sont optimales pour des ensembles de tâches indépendantes synchrones à échéances implicites sur un système multiprocesseur. Aucun autre résultat n'est connu à ce jour en multiprocesseur. Comme on peut le constater, le cas de tâches dépendantes n'est quasiment pas traité.

2.3 Analyse d'ordonnançabilité

L'analyse d'ordonnançabilité a pour but de valider formellement que l'exécution d'un ensemble de tâches sur une architecture avec une politique donnée respecte toutes les contraintes

temporelles et de précédence. Cette activité est donc fondamentale pour les systèmes temps réels critiques.

2.3.1 Deux approches : méthodes analytiques et méthodes de parcours exhaustif

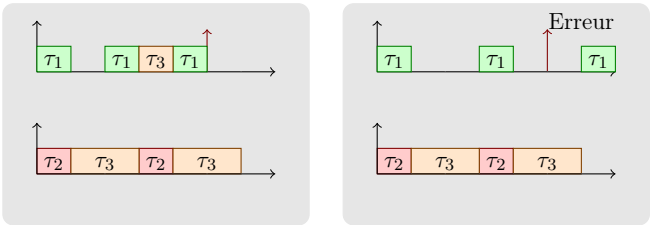
Dans cette section nous commençons par l'illustration de deux comportements contre-intuitifs pour mettre en relief la difficulté de l'analyse. Ensuite nous présentons deux familles de méthodes pour la réalisation d'une analyse d'ordonnançabilité : les *méthodes analytiques* et *les méthodes exactes* de parcours exhaustif.

Deux comportements contre-intuitifs

L'analyse d'ordonnançabilité multiprocesseur doit faire face à des problèmes inexistants en monoprocesseur dû à deux comportements contre-intuitifs. Graham [Gra69] montre l'existence d'*anomalies du comportement* : soit un ensemble de tâches ordonnançable avec une politique d'ordonnancement, il se peut qu'en réduisant sa charge sur le système (en diminuant les pires temps d'exécution ou en augmentant les échéances), le système devienne non ordonnançable. La figure 2.13 illustre ce résultat. Dans la partie gauche de la figure 2.13(b) l'ensemble est ordonnançable en gRM par contre si on augmente la période de τ_1 de 2 à 3, l'ensemble devient non ordonnançable.

τ_i	O_i	T_i	D_i	C_i
τ_1	0	2	1	1
τ_2	0	3	1	1
τ_3	0	6	6	5

(a)



(b)

FIGURE 2.13 – Illustration d'une anomalie [Gra69]

Dans les systèmes monoprocesseur, le scénario pire cas est le démarrage synchrone de toutes les tâches, et on parle alors d'*instant critique*. Lauzat et al [LMM98] ont montré que dans le cas multiprocesseur cette propriété n'est pas conservée comme illustré dans l'image 2.14(b). Le pire temps de réponse (définie dans la section 2.1) de la tâche τ_4 n'est pas au moment où toutes les tâches démarrent simultanément.

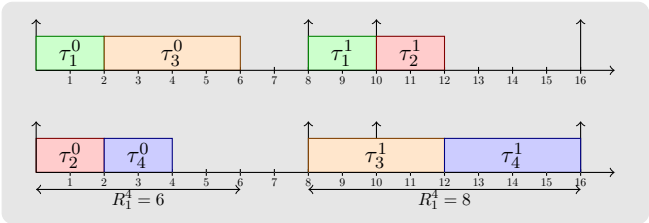
Méthodes analytiques

Les méthodes analytiques essaient de vérifier l'ordonnançabilité d'un ensemble de tâches à base de formules *Nécessaires et Suffisantes (CNS)* ou simplement *Suffisantes (CS)*. Les premières impliquent que tous les ensembles de tâches ordonnançables vérifient la condition. Par contre, les deuxièmes impliquent que si un ensemble vérifie la condition il est ordonnançable, sinon on ne peut pas conclure.

Dans le cas multiprocesseur, les conditions existantes sont essentiellement CS. Les bornes sont traditionnellement exprimées en terme de charge du système, $\delta_{sum} = \sum_{\tau_i \in \mathcal{S}}(C_i/D_i)$ et pour

τ_i	O_i	T_i	D_i	C_i
τ_1	0	8	8	2
τ_2	0	10	10	2
τ_3	0	8	8	4
τ_3	0	8	8	4

(a)



(b)

FIGURE 2.14 – Illustration de l'instant critique

chaque politique en particulier : un ensemble de tâches S est ordonnançable pour une politique P si δ_{sum} est inférieur ou égal à une borne B.

Il existe de nombreux articles proposant des critères suffisants pour les politiques FP et gEDF. A titre d'exemple, nous donnons un critère suffisant en priorité fixe.

Théorème 1. *Bertogna et al. [BCL09] présentent un critère suffisant pour des ensembles de tâches synchrones à échéances contraintes et pour un algorithme à priorité fixe. Si toutes les tâches respectent l'équation 2.4, l'ensemble est ordonnançable :*

$$\sum_{i<k} \min(\mathcal{W}_i(D_k), D_k - C_k + 1) < m(D_k - C_k + 1) \tag{2.4}$$

où $N_{i,k} = \left\lfloor \frac{L + D_i - C_i}{T_i} \right\rfloor + 1$ *et* $\mathcal{W}_i(L) = N_i(L)C_i + \min(C_i, L + D_i - C_i - N_i(L)T_i))$.

Exemple 10. *Considérons l'ensemble de tâches* $\mathcal{S} = \{\tau_1 = (0, 6, 4, 4), \tau_2 = (0, 6, 4, 2), \tau_3 = (0, 8, 7, 4)\}$. *On applique les équations 2.4 sur une architecture à deux processeurs. On peut vérifier que*

$$\begin{aligned}
N_1(D_2) &= \left\lfloor \tfrac{D_2+D_1-C_1}{T_1} \right\rfloor + 1 = 1 \\
N_1(D_3) &= \left\lfloor \tfrac{D_3+D_1-C_1}{T_1} \right\rfloor + 1 = 2 \\
N_2(D_3) &= \left\lfloor \tfrac{D_3+D_2-C_2}{T_2} \right\rfloor + 1 = 2
\end{aligned}$$

$$\begin{aligned}
W_1(D_2) = \;& N_1(D-2)C_1 + \min(C_1, D_2 + D_1 - C_1 - N_1(D_2)T_1)) = \\
& 1 \times 4 + \min(4, 4 + 4 - 4 - 1 \times 6) = 2 \\
W_1(D_3) = \;& N_1(D-3)C_1 + \min(C_1, D_3 + D_1 - C_1 - N_1(D_3)T_1)) = \\
& 2 \times 4 + \min(4, 7 + 4 - 4 - 2 \times 6) = 3 \\
W_2(D_3) = \;& N_2(D-3)C_2 + \min(C_2, D_3 + D_2 - C_2 - N_2(D_3)T_2)) = \\
& 2 \times 2 + \min(2, 7 + 4 - 2 - 2 \times 6) = 1
\end{aligned}$$

Alors, l'ensemble est ordonnançable car

$$\begin{aligned}
&\min(W_1(D_2), D_2 - C_2 + 1) && < m(D_2 - C_2 + 1) \\
&\Rightarrow \min(2, 4 - 2 + 1) && < 2(4 - 2 + 1) \Rightarrow 2 < 6
\end{aligned}$$

$$\begin{aligned}
&\min(W_1(D_3), D_3 - C_3 + 1) + \min(W_2(D_3), D_3 - C_3 + 2) && < m(D_3 - C_3 + 1) \\
&\Rightarrow \min(3, 7 - 4 + 1) + \min(1, 7 - 4 + 1) && < 2(7 - 4 + 1) \Rightarrow 4 < 8
\end{aligned}$$

Les méthodes analytiques sont souvent pessimistes et ne concluent pas sur certains ensembles de tâches ordonnançables. En outre, à notre connaissance, il n'existe pas d'analyse pour gLLF ou pour des ensembles de tâches avec des contraintes de précédence. Pour la politique LLREF les résultats concernent les ensembles de tâches synchrones à échéances implicites, pour lesquels cette politique est optimale. Par contre, il n'existe pas de bornes d'ordonnançabilité dans le cas asynchrone ou à échéance contrainte.

Méthodes exactes à base d'exploration exhaustive

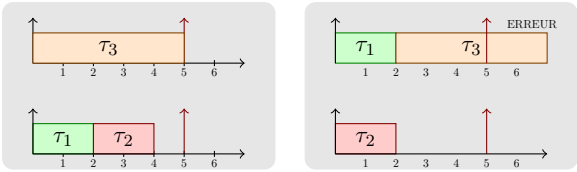
La solution par calcul de bornes, exposée dans la section 2.3.1, a une bonne complexité de calcul. En revanche, les bornes sont souvent pessimistes ce qui amène à un sur-dimensionnement du système. Pour éviter cette perte de performances et pour traiter les cas non pris en compte par les conditions suffisantes, plusieurs auteurs ont travaillé sur des techniques d'exploration exhaustive. Une méthode de parcours exhaustif consiste à générer toutes les séquences d'ordonnancement obtenues avec une politique d'ordonnancement donnée afin de vérifier si toutes les traces respectent les contraintes temporelles et de précédence. Pour appliquer cette approche, il faut considérer trois aspects particuliers : l'*indéterminisme*, l'*intervalle de faisabilité* et la *prédictibilité*.

Indéterminisme Certaines politiques d'ordonnancement, comme par exemple gEDF ou LLREF, peuvent produire une exécution du système non déterministe : pour une même situation, la politique d'ordonnancement peut prendre plusieurs décisions. Selon les décisions prises, la simulation générée peut être ordonnançable ou non.

Exemple 11. *Prenons l'ensemble de tâches de 2.15(a). On réalise une simulation sur deux processeurs avec gEDF. Puisque les échéances de toutes les tâches à l'instant 0 sont similaires, toutes les tâches sont éligibles pour être exécutées. Comme il n'y a que deux processeurs plusieurs exécutions sont possibles, autant que de permutations entre les tâches. Dans le premier diagramme de Gantt, on affecte les deux priorités les plus élevées à τ_1 et τ_3 et la simulation est ordonnançable. Par contre, dans le deuxième diagramme de Gantt, on affecte les deux priorités les plus élevées à τ_1 et τ_2, et τ_3 ne respecte pas son échéance.*

τ_i	O_i	T_i	D_i	C_i
τ_1	0	5	5	5
τ_2	0	5	5	2
τ_3	0	5	5	2

(a)



(b)

FIGURE 2.15 – Exemple de simulation

L'indéterminisme est du à la possibilité d'affecter la même priorité à plusieurs tâches à un instant t. Dans cette situation l'ordonnanceur peut, en théorie, choisir n'importe quel ordre d'affectation entre les tâches avec la même priorité et en conséquence, obtenir plusieurs séquences d'ordonnancement. Dans la pratique, les ordonnanceurs sont programmés de manière déterministe. Il suffit donc de simuler et analyser le comportement exact de l'ordonnanceur.

Intervalle de faisabilité Afin de mener correctement une exploration exhaustive, il faut visiter tous les états accessibles pour déterminer que tous sont corrects. Pour que l'exploration soit finie

il faut soit que l'ensemble des états soit fini, soit qu'il existe une représentation finie de l'ensemble infini. Une exécution temps réel périodique est par définition infinie mais comme les paramètres d'ordonnancement sont bornés, une exécution repasse nécessairement à un instant t_r dans un état équivalent à un état déjà visité à un instant t_0. Si la politique utilisée est déterministe et sans mémoire, les décisions à partir t_r sont les mêmes qu'à partir de t_0. On peut conclure que l'exécution se réalise cycliquement entre $[t_0..t_r]$. Si entre $[t_0..t_r]$ toutes les contraintes temporelles et de précédence sont respectées, on peut dire que l'exécution cyclique de $[t_0..t_r]$ respectera toujours les contraintes temporelles. Dans ce cas, on dit que $[t_0..t_r]$ est l'*intervalle de faisabilité* de $\mathcal{S}$. En conséquence, il suffit de simuler le(s) comportement(s) du système entre ces deux valeurs pour assurer l'ordonnançabilité. Cependant, le problème est de trouver la valeur de t_0 et t_r.

Cette question est résolue dans le cas monoprocesseur [LM80] où l'intervalle de faisabilité se trouve dans l'intervalle $[0, H]$ pour les tâches synchrones et $[0, \max O_i + 2H]$ pour les tâches asynchrones. Dans le cas multiprocesseur, Cucu et Goossens ont exposé plusieurs résultats concernant cette question dans deux articles de conférence [CG06, CG07] et un troisième plus complet de revue [CGG11]. Ils montrent que pour toute politique d'ordonnancement déterministe et conservative l'intervalle est fini, ce qui est équivalent à affirmer que l'exécution est cyclique. Néanmoins, sa cyclicité varie selon les dates de démarrage des tâches.

Pour un ensemble de tâches synchrones à échéances contraintes, l'intervalle de faisabilité est $[0, H)$ [CG06] . Dans le cas asynchrone, les résultats sont plus limités. Dans [CG06, CGG11], Cucu et Goossens montrent que dans le cas d'un ordonnanceur à priorité fixe la fenêtre de répétition est l'intervalle $[S_n, S_n + H]$ où S_n est calculé de manière inductive :

$$\begin{cases} S_1 &= O_1 \\ S_i &= \max\left\{O_i, O_i + \left\lceil \frac{S_{i-1}-O_i}{T_i} \right\rceil T_i\right\}, \forall i \in \{2, 3, ..., n\} \end{cases} \tag{2.5}$$

Exemple 12. *Prenons l'ensemble de tâches suivant :*

τ_i	O_i	T_i	D_i	C_i
τ_1	2	8	8	4
τ_2	5	5	5	3
τ_3	0	20	20	12

$$S_1 = 2$$
$$S_2 = \max\left\{5, 5 + \left\lceil \frac{2-5}{5} \right\rceil 5\right\} = 5$$
$$S_3 = \max\left\{0, 0 + \left\lceil \frac{5-0}{20} \right\rceil 20\right\} = 20$$

En appliquant la formule 2.5, puisque que l'hyper-période est $H = 40$, l'intervalle de répétition est $[20, 60]$ pour une politique à priorité fixe.

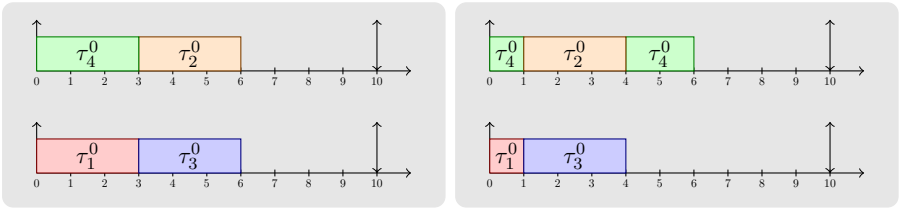
Cependant, pour des ordonnanceurs à priorité dynamique la fenêtre est inconnue. Nous avons borné l'intervalle de faisabilité dans cette thèse, ce qui sera détaillé dans la section 4.2.2. Notre borne est extrêmement pessimiste.

Prédictibilité La troisième question concerne la validité de l'analyse sur le système réel. En effet, le parcours se fait avec des temps d'exécution égaux aux WCET. Dans la réalité, il se peut que la tâche utilise moins de temps CPU. Il faut alors garantir que le système reste ordonnançable quelle que soit la variation des temps d'exécution. Ha et Liu [HL93] ont appelé cette propriété la *prédictibilité*. Une politique d'ordonnancement est prédictible si en réduisant le temps d'exécution des jobs, les temps de réponse n'augmentent pas. Cette propriété est valide pour tous les ordonnanceurs en-ligne considérés dans la thèse lorsque les tâches sont indépendantes. Ce n'est pas le cas lorsqu'il y a des précédences gérées par des mécanismes de synchronisation.

Exemple 13. *Considérons l'exemple suivant s'exécutant sur deux processeurs avec la politique FP :*

τ_i	O_i	T_i	D_i	C_i
τ_1	0	10	3	3
τ_2	0	10	10	3
τ_3	0	10	10	3
τ_4	0	10	5	3

$$\mathcal{R}$$
$$\tau_1 \rightarrow \tau_2$$
$$\tau_1 \rightarrow \tau_3$$



(a) (b)

FIGURE 2.16 – Exemple de non prédictibilité

Dans la première exécution, illustrée dans l'image 2.16(a), le temps de réponse de τ_4 est de 3. Par contre, dans la figure 2.16(b), la tâche τ_1 réduit son temps d'exécution de 2 et le temps de réponse de τ_4 augmente à 6, rendant le système non ordonnançable.

2.3.2 Algorithmes existants de parcours exhaustif

Plusieurs auteurs ont proposé des méthodes de recherche pour l'exploration de toutes les simulations possibles et ainsi analyser d'une façon exacte l'ordonnançabilité du système.

Recherche par automates temporisés

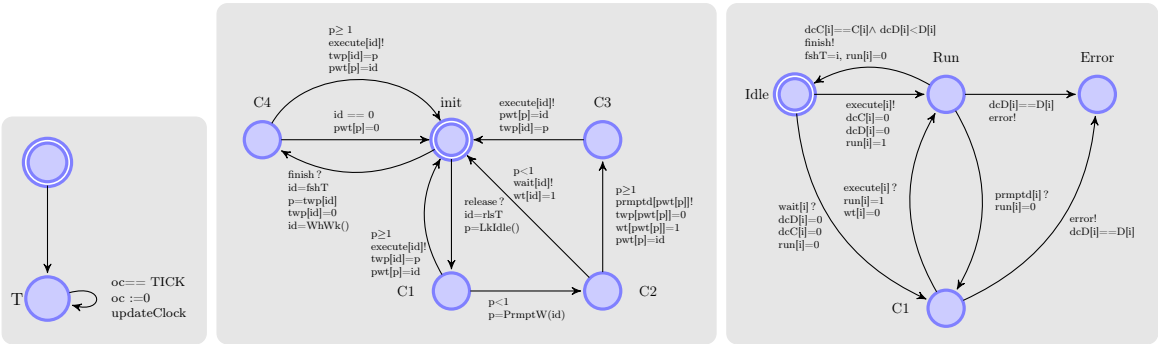
Une première solution repose sur l'utilisation d'automates temporisés. L'outil TIMES [2] est le pionnier [AFM+02] à vérifier l'ordonnançabilité avec le model checker UPPAAL pour des systèmes monoprocesseur. D'autres auteurs ont proposé des solutions du même type pour des architectures multiprocesseur, comme par exemple, Guan et al. [GGD+07, GGL+08], David et al. [DILS10] et Fersman et al. [FY04, FKPY07]. Nous expliquons plus en détail les deux premières solutions.

Guan et al. [GGD+07] modélisent le problème d'analyse d'ordonnançabilité pour une politique à priorité fixe comme un problème de model checking. Le modèle est codé par des automates temporisés manipulés à l'aide de l'outil UPPAAL. Les ensembles de tâches sont limités aux ensembles synchrones et aux échéances implicites. L'ordonnancement à priorité fixe est préemptif et avec migration complète (ordonnancement global). L'illustration 2.17 montre les automates des trois entités du système. L'horloge imprime le rythme de la vérification. A chaque tirage de la boucle, la fonction `updateClock` s'exécute et l'automate de l'ordonnanceur peut mettre à jour l'état de chaque tâche. Une tâche est représentée par 4 états :

- `idle` : la tâche n'est pas active. C'est l'état initial de toutes les tâches ;
- `run` : la tâche est en exécution sur un processeur ;

2. Disponible sur le site http://www.timestool.com/

– `wait` : la tâche a été préemptée par une tâche plus prioritaire et attend que l'ordonnanceur l'affecte à un processeur ;
– `error` : la tâche a raté une contrainte temporelle.



(a) Horloge (b) Ordonnanceur (c) Tâche

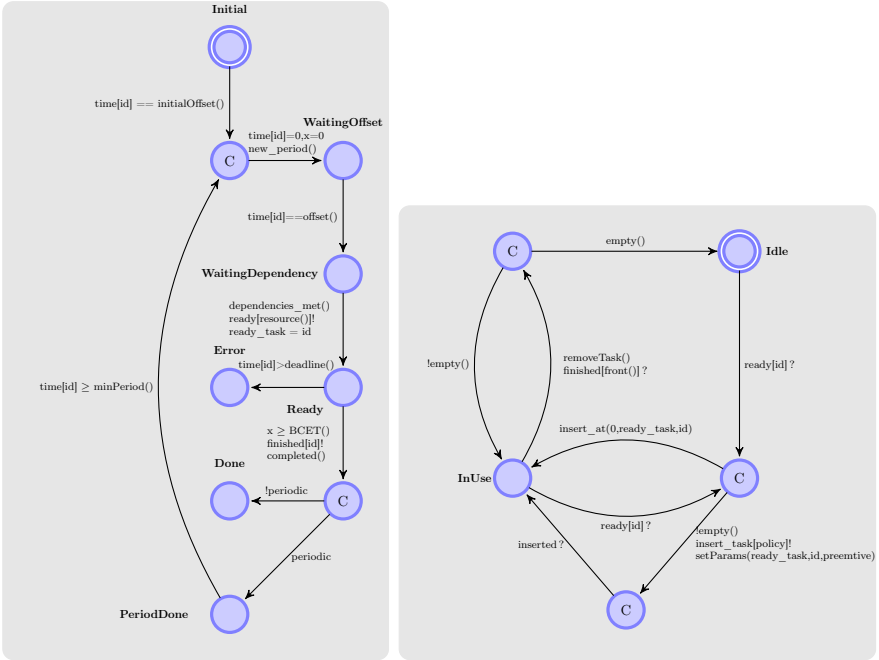
FIGURE 2.17 – Guan et al. modèles

Enfin l'ordonnanceur gère le changement d'état de chaque tâche guidé par la politique d'ordonnancement. Un ordonnancement est correct si on vérifie qu'aucune tâche n'arrive jamais à l'état "error".

David et al. utilisent les mêmes idées et proposent dans [DILS10] un framework dédié à l'analyse d'ordonnançabilité sur multiprocesseur. Le modèle de tâches est plus riche : précédences entre les tâches, incertitude de la date d'arrivée d'une tâche et modélisation du temps d'exécution avec le BCET (Best Case Execution Time) et le WCET (Worst Case Execution Time). Le framework permet d'analyser des ensembles des tâches avec des politiques préemptives et non préemptives. Également, la migration peut être ou non permise. De nouveau l'outillage repose sur l'utilisation d'UPPAAL. L'illustration 2.18 montre les deux automates temporisés génériques. L'automate de tâche 2.18(a) s'applique aux tâches périodiques si le booléen *Periodic* est à vrai, sinon il s'agit de jobs. Cinq états forment sa structure :

– `initial` : pour passer à l'état suivant la tâche doit attendre à sa date de réveil ;
– `waiting` : cette étape est formée de deux états, un pour attendre le temps de démarrage d'une période et un deuxième pour attendre si la tâche est contrainte par une précédence ;
– `ready` : soit la tâche est en exécution soit la tâche est en attente pour la libération d'un processeur ;
– `error` : la tâche n'a pas respecté son échéance ;
– `done` : la tâche a correctement terminé l'exécution de son job. A cette étape, soit la tâche n'est pas périodique, alors la transition l'amène vers un état final, *Done*, soit elle est périodique et elle recommence un nouveau job.

L'automate du processeur prend en entrée trois paramètres : l'identificateur du processeur, le fait que la préemption soit autorisée et la politique utilisée. Les ordonnanceurs proposés sont : FIFO, priorité fixe et gEDF. Deux états principaux décrivent la situation d'un processeur, *idle* et *inUse*. Un processeur attend une tâche à exécuter dans l'état *idle*. Un processeur exécutant une tâche se trouve dans *inUse*. Si une tâche demande un processeur car elle commence une exécution, deux chemins sont possibles : le processeur est libre alors elle peut l'occuper ou le processeur n'est pas libre alors elle doit attendre. L'activation est réalisée par la politique d'ordonnancement qui décide quelles tâches ont la priorité la plus élevée. Lorsqu'une tâche termine son exécution, elle libère le processeur qui passe à *idle*.



(a) Tâche (b) Processeur

FIGURE 2.18 – Modèle de David et al. (1)

L'automate de la politique d'ordonnancement gEDF est montré dans la figure 2.19. Il est composé de trois états. La première transition lit les paramètres d'ordonnancement et initialise la variable `place` : 0 si l'ordonnanceur est préemptif, la valeur du nombre de tâches candidates à cette ressource si l'ordonnanceur n'est pas préemptif. Pour un ordonnanceur préemptif, on parcourt toute la liste de tâches candidates, sur la boucle du deuxième état, pour chercher cela avec l'échéance la plus petite. La transition pour aller au troisième état met la tâche avec l'échéance la plus petite dans `place` (dans le cas non préemptif on laisse la tâche en exécution) et on l'affecte au processeur avec `insert_at(place,tid,rid)`. On revient à l'état initial et on prévient le processeur qu'on a fini l'affectation de tâche pour qu'il passe à l'état `inUse`. Pour vérifier que l'ensemble de tâches est ordonnançable, il faut vérifier qu'aucune tâche n'arrive à l'état `error`. Autrement dit, $A[]\ forall(i:n)\ not\ Task(i).Error$.

Par rapport aux performances, Guan et al. font une évaluation en se comparant aux méthodes analytiques. Pour 200 ensembles des tâches synchrones ils trouvent 154 solutions, avec le test de Baker [Bak06] 64 et avec le test de Anderson [ABJ01] 98. Malgré les bons résultats, les tests ont été réalisés sur des ensembles de tâches très petits (5 tâches). Guan et al. et David et al. rencontrent le problème de l'explosion combinatoire, ils traitent respectivement une trentaine de tâches et une cinquantaine de tâches.

Recherche par réseaux de Petri temporisés

Dans [LR09], les auteurs proposent une méthode de vérification des aspects temporels pour un modèle de tâches asynchrones et une politique d'ordonnancement (gEDF ou priorité fixe) préemptive et à migration non permise. Le modèle du système peut aussi contenir des sémaphores, des réseaux CAN, des événements... La modélisation proposée se fait avec des réseaux de Petri temporisés (TPN) qui représentent l'accès concurrent des tâches aux processeurs. L'analyse se

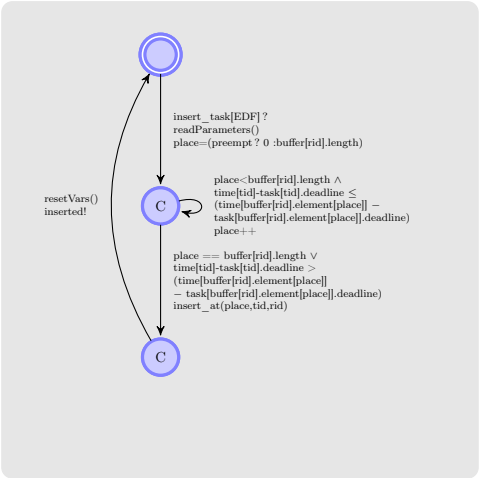


FIGURE 2.19 – Automates de politiques

fait en deux étapes : un calcul de l'espace d'état du TPN pour obtenir un automate hybride linaire (LHA) équivalent et une analyse exacte sur le LHA avec le model checker HYTECH [3].

Exemple 14. *Reprenons l'exemple présenté dans [LR09]. La figure 2.20 représente le TPN correspond à l'exécution de deux tâches, τ_1 et τ_2, sur un processeur avec un ordonnanceur EDF avec $\mathcal{S} = \{\tau_1 = (1, 10, 10, 1), \tau_2 = (1, 8, 10, 1)\}$. La périodicité est obtenue grâce à la transition t_1 qui libère un jeton toutes les 10 unités de temps. La fonction γ affecte une tâche à la place. Si $\gamma = \emptyset$ la place n'est pas affectée à une tâche. Les valeurs de δ correspondent aux échéances de chaque tâche. Le fonctions $B(\tau)$ et $E(\tau)$ indiquent respectivement la transition de début et de fin de la tâche. La fonction $\Pi(\tau)$ affecte une tâche à un processeur. Dans l'exemple, un seul processeur est utilisé, en conséquence $\Pi(\tau_1) = \Pi(\tau_2)$. En dernier, la fonction $sched(\Pi(\tau))$ affecte la priorité des tâches selon la politique.*

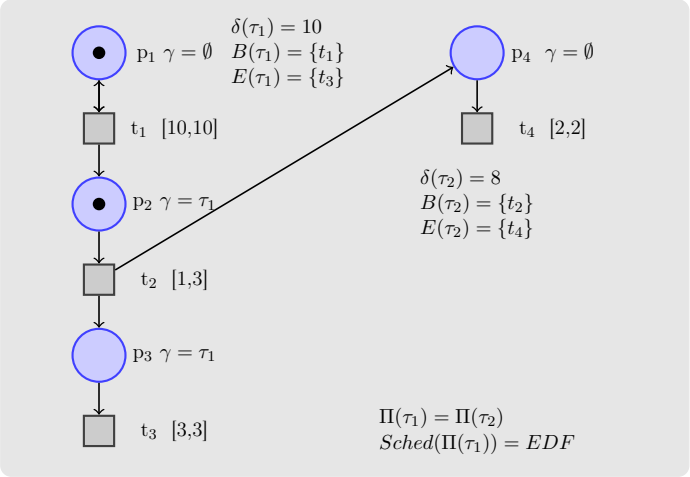


FIGURE 2.20 – Modélisation TPN de deux tâches sur un processeur

Pour l'analyse de ce modèle, le TPN est d'abord traduit en automate hybride linaire et analysé

3. disponible sur le site http://embedded.eecs.berkeley.edu/research/hytech/

ensuite avec HyTech.

Les auteurs présentent une étude de performances pour estimer la limite de la méthode proposée et la comparer avec une modélisation directe en LHA. Les modèles étudiés sont composés de plusieurs processeurs et d'un réseau de communication CAN reliant les processeurs. Les résultats montrent que la méthode arrive à vérifier la correction d'ensembles de 11 tâches qui s'exécutent sur une architecture à 7 processeurs reliés par un réseau CAN.

Recherche par automate fini

Guan et al. [GGL+08] proposent une modélisation à base d'automates finis et une analyse avec le model checker NuSMV[4]. Le modèle de tâches choisi est synchrone avec échéances implicites sans contraintes de précédence et les politiques modélisées sont FP et gEDF avec préemption et migration permise.

Exemple 15. *Pour l'ensemble de tâches extrait de [GGL+08] et une politique à priorité fixe :*

τ_i	D_i	C_i	*prio*
τ_1	6	2	1
τ_2	8	3	2
τ_3	10	2	3
τ_3	12	4	4

```
MODULE main
VAR
   c1:0..2; c2:0..3; c3:0..2; c4:0..4;
   d1:0..6; d2:0..8; d3:0..10 d4:0..12;
DEFINE
   C1:=2; C2:=3; C3:=2; C4:=4;
   D1:=6; D2:=8; D3:=10; D4:=12;
   PN:=2;
   T1_rq := c1 < C1;
   T1_lv := c1 < C1 & d1 < D1 - 1;
   T1_lv_rls := c1 = C1 - 1 & d1 = D1 - 1;
   T1_slp := c1 = C1 & d1 < D1 - 1;
   T1_slp_rls := c1 = C1 & d1 = D1 - 1;
   T1_miss := D1 - d1 < C1 - c1;
  --Similarly for tasks 2, 3 and 4.
   T1_gp:=T1_rq;
   T2_gp:=T2_rq & (T1_rq < PN);
   T3_gp:=T3_rq & (T1_rq+T2_rq < PN);
   T4_gp:=T4_rq & (T1_rq+T2_rq+T3_rq < PN);
ASSIGN
   init(c1):=2; init(c2):=3;
   init(c3):=2; init(c4):=4;
   init(d1):=0..5; init(d2):=0..7;
   init(d3):=0..9; init(d4):=0..11;
   next(c1) := case
      T1_lv & T1_hp : c1 + 1;
      T1_lv_rls & T1_hp | T1_slp_rls: 0;
      T1_slp | !T1_hp : c1;
      1 : 0;
   esac;
   next(d1) := (d1 + 1) mod D1;
  -- similarly for tasks 2, 3 and 4.
DEFINE
```

4. model checker symbolique disponible sur http://nusmv.fbk.eu/

```
MISS:= T1_miss|T2_miss|T3_miss|T4_miss;
SPEC AG !MISS
```

La première partie définit les variables qui modélisent l'état de chaque tâche : c_i correspond au temps d'exécution restant jusqu'à la prochaine activation de τ_i et d_i le temps restant jusqu'à la prochaine échéance de τ_i. Ensuite, on déclare les valeurs constantes des paramètres de tâches (C_i et D_i). Le nombre de processeurs est codé par la constante PN. Une tâche évolue de la manière suivante :

- *la condition $T1_rq := c1 < C1$ indique que la tâche est active et n'a pas terminé son exécution ;*
- *la condition $T1_lv := c1 < C1$ & $d1 < D1 - 1$ indique que la tâche est active et il reste plus d'une unité de temps pour la finalisation du job courant ;*
- *la condition $T1_lv_rls := c1 = C1 - 1$ & $d1 = D1 - 1$ indique que la tâche est active et termine le job dans une unité de temps ;*
- *la condition $T1_slp := c1 = C1$ & $d1 < D1 - 1$ indique que la tâche est finie et le nouveau job ne se réveille pas au prochain instant ;*
- *la condition $T1_slp_rls := c1 = C1$ & $d1 = D1 - 1$ indique que la tâche est finie et le prochain job se réveille dans une unité de temps ;*
- *la condition $T1_miss := D1 - d1 < C1 - c1$ indique que la tâche n'est pas ordonnançable car le temps d'exécution restant est plus grand que le temps jusqu'à l'échéance du job.*

Enfin, T_i_gp indique si une tâche a l'accès à un processeur. Pour cela on compte le nombre de tâches plus prioritaires que τ_i avec T_i_rq (les tâches sont ordonnées par ordre décroissant de priorité). Si le nombre de tâches plus prioritaires est plus petit que le nombre de processeurs, τ_i peut s'exécuter.

Dans la partie `assign`*, on initialise toutes les variables avec la valeur des paramètres de tâches. Ensuite on définit les fonctions de mise à jour pour les variables c_i et d_i (on note $c_i(t)$ la valeur de c_i à l'instant t) :*

$$c_i(t+1) = \begin{cases} c_i(t) + 1 & \text{si } T_i_lv \wedge T_i_hp \\ 0 & \text{si } T_i_lv_rls \wedge T_i_hp \vee T1_slp_rls \\ c_i(t) & \text{si } T_i_slp \wedge \neg T_i_hp \end{cases}$$

et $d_i(t+1) := (d_i(t) + 1) \ mod \ D_i$
Enfin, on a la définition de la formule à vérifier : $MISS$ est vrai si toutes les tâches respectent leurs échéances. On doit donc vérifier que $AG!MISS$.

Cette approche est plus efficace que les approches par automates temporisés. Les auteurs peuvent traiter des ensembles constitués d'une cinquantaine de tâches en FP et une trentaine en gEDF. Ils se comparent également avec des conditions suffisantes et sont moins pessimistes. Pour un ordonnanceur FP ils trouvent que les ensembles sont ordonnançables jusqu'à $\approx 80\%$ de occupation du processeur ($\approx 50\%$ pour les méthodes analytiques) et $\approx 90\%$ en gEDF ($\approx 60\%$ pour les méthodes analytiques).

Algorithmes *ad-hoc* de recherche exhaustive

Baker et Cirinei présentent dans [BC07] un algorithme d'exploration exhaustive pour une analyse exacte d'ordonnançabilité avec des politiques FP, gEDF et gLLF et un modèle de tâches sporadique $\mathcal{S} = (T, D, C)$. La modélisation propose d'encoder l'état du système S avec le tuple :

$$\langle((nat(S_1), rct(S_1)), \cdots, (nat(S_n), rct(S_n))\rangle$$

La valeur de $nat(S_i)$ indique la prochaine date d'arrivée de la tâche τ_i et $rct(S_i)$ le temps d'exécution restant du job de τ_i. Ils ont défini un ensemble $run(S)$ avec toutes les tâches qui doivent être exécutées, c'est-à-dire toutes les tâches dont $rct(S_i) > 0$. Le système commence avec tous les paramètres à zéro $((0,0), \cdots, (0,0))$. L'évolution du système se fait à travers deux types de transitions :

- *transition clock-tick* : cette transition fait avancer le temps et met à jour l'état des tâches. Pour cela, les auteurs définissent la fonction successeur $S' = Next(S)$:

$$rtc(S'_i) = \begin{cases} rct(S_i) & \text{si } \tau_i \notin run(S) \\ (rct(S_i) - 1) & \text{si } \tau_i \in run(S) \end{cases}$$

$$nat(S'_i) = \begin{cases} \max(0, nat(S_i) - 1) & \text{si } rct(S_i) = 0 \\ nat(S_i) - 1 & \text{si } rct(S_i) > 0 \end{cases}$$

- *transition ready* : l'ensemble *ready(S)* est composé des tâches respectant $rct(S'_i) = 0$ et $nat(S'_i) = 0$. La transition ready est tirable si l'ensemble *ready(S)* n'est pas vide et à chaque fois que cette transition est lancée un job est activé :

$$rct(S'_i) = C_i, nat(S_i) + D_i \leq nat(S'_i) \leq D_i$$

Exemple 16. *Soit l'ensemble de tâches* $\mathcal{S} = \{\tau_1 = (6,4,4), \tau_2 = (6,4,2), \tau_3 = (8,7,4)\}$, *deux processeurs et une politique d'ordonnancement gEDF déterministe. L'exécution est montrée dans la figure 2.21.*

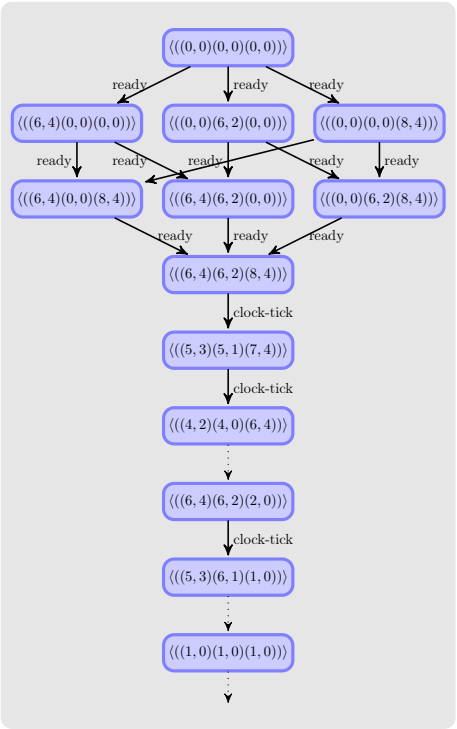


FIGURE 2.21 – Dépliage selon l'algorithme de Baker et Cirinei

Au début de l'exécution toutes les tâches veulent s'activer. Elles démarrent avec la transition ready *en cascade comme illustré sur l'image. On est obligé de démarrer toutes le tâches avant de*

commencer l'évolution temporelle. Pour cela, à la fin de l'activation tous les chemins convergent vers le même état. Une fois que toutes les tâches sont actives le système avance avec des transitions clock-tick*. On peut remarquer que l'avancement à travers les transition* clock-tick *se fait en séquence, c'est-à-dire pour toute transition* clock-tick *on a un seul état de sortie. Cela est dû à l'hypothèse d'utiliser un ordonnanceur déterministe, ce qui implique qu'il n'y a qu'un seul choix d'exécution.*

Les auteurs proposent un algorithme force brute de parcours complet de l'espace d'état. Cette solution, ne voulant être que la première recherche exhaustive complète, rencontre rapidement le problème de l'explosion combinatoire. Une amélioration par des techniques d'anti-chaîne est proposée en [LGG11].

Cucu et Goossens [CGG11] proposent une autre modélisation du système pour un ordonnanceur gEDF pour des ensembles de tâches périodiques, asynchrones à échéances arbitraires. D'abord ils proposent une modélisation de l'état du système par rapport au temps $\theta(t)$ composé de l'état de chacune de tâches $\theta_i(t)$:

$$\theta_i(t) = \begin{cases} (-1, x, 0) & \text{si aucun job de } \tau_i \text{ n'a été activé avant ou en } t \\ & \text{x est le temps restant jusqu'à l'activation du premier job ;} \\ (y, z, u) & \text{y est le nombre de jobs de } \tau_i \text{ actifs à t. z est le temps écoulé à t depuis} \\ & \text{l'arrivée du plus ancien job de } \tau_i\text{. u est le temps d'exécution restant} \\ & \text{du plus ancien job } (u = 0) \text{ si aucun job n'est actif t} \end{cases}$$

Ils modélisent aussi chaque processeur à un instant t par $\sigma_j(t)$.

$$\sigma_j(t) = \begin{cases} 0, & \text{s'il n'y a pas de tâches en exécution sur le processeur j à t ;} \\ i, & \text{si la tâche i est en exécution sur le processeur j à t.} \end{cases}$$

Ils proposent ensuite un parcours par simulation (algorithme 2.1) où à des instants périodiques on contrôle si on est arrivé à la partie périodique de l'ordonnancement. La fenêtre pour ce type d'ordonnanceur est inconnue mais sa largeur est un multiple de H à partir de $\forall i = 0..n, \max(O_i)$.

Fonction 2.1 : EDF

Entrées : ensemble de tâches $\mathcal{S}$
Sorties : faisable

```
1  Schedule (dès O) à O_max;
2  S_1 ← θ(O_max);
3  Schedule (dès O_max à O_max + P;
4  S_2 ← θ(O_max + P);
5  current_time ← (O_max + P);
6  nbtaskawake_inter ⟵ 0;
7  tant que S_1 ≠ S_2 faire
8      S_1 ← S_2;
9      Schedule (dès current_time) à current_time +P;
10     current_time← current_time +P;
11     S_2 ← θ(current_time);
12 retourner true
```

Exemple 17. *Soit l'ensemble de tâches* $\mathcal{S} = \{\tau_1 = (6, 4, 4), \tau_2 = (6, 4, 2), \tau_3 = (8, 7, 4)\}$ *exécuté sur deux processeurs et une politique d'ordonnancement gEDF déterministe alors* $\theta(t)$ *est :*

$$
\begin{array}{ll}
\text{instant } 0 \rightarrow & ((1,0,0)(1,0,0)(1,0,0)) \\
\text{instant } 1 \rightarrow & ((1,1,1)(1,1,1)(1,1,0)) \\
\text{instant } 2 \rightarrow & ((1,2,2)(1,2,2)(1,2,0)) \\
\text{instant } 3 \rightarrow & ((1,3,3)(0,3,0)(1,3,1)) \\
\ldots & \ldots \\
\text{instant } 6 \rightarrow & ((1,0,0)(1,0,0)(0,6,0)) \\
\text{instant } 7 \rightarrow & ((1,1,1)(1,1,1)(0,7,0)) \\
\text{instant } 8 \rightarrow & ((1,2,2)(1,2,2)(1,0,0)) \\
\ldots & \ldots \\
\text{instant } 23 \rightarrow & ((0,5,0)(0,5,0)(0,7,0)) \\
\text{instant } 24 \rightarrow & ((1,0,0)(1,0,0)(1,0,0))
\end{array}
$$

Les états parcourus sont au nombre de 24 mais on stocke uniquement 2, $S_0 = 0$ *et* $S_1 = 24$.

2.3.3 Ordonnancement optimal hors-ligne

Une politique d'ordonnancement en-ligne nécessite de prendre des décisions d'ordonnancement au moment de l'exécution. Une autre solution possible consiste à choisir un ordonnancement hors-ligne, c'est-à-dire calculer une séquence finie qu'on exécute sur la plateforme en la répétant indéfiniment. Cette solution offre plusieurs propriétés intéressantes : (1) *prédictibilité*, les coûts des préemptions, migrations et changements de contexte sont connus avant l'exécution du système ; (2) *simplicité*, l'utilisation de sémaphores ou autres mécanismes de synchronisation n'est pas nécessaire. Cela simplifie la plateforme sur laquelle les tâches et la politique d'ordonnancement doivent être exécutées ; (3) *efficacité*, l'exécution d'une séquence obtenue hors-ligne est très simple car aucune décision ne doit être prise au moment de l'exécution. Plusieurs auteurs dans le domaine des SoC (System on Chip) conseillent l'utilisation de ce type de techniques [CYC+05] ; (4) *optimalité*, une recherche de séquence hors-ligne peut théoriquement toujours trouver un ordonnancement valide s'il en existe un, contrairement aux ordonnanceurs en-ligne [HL88]. Cette propriété est possible grâce à deux propriétés conservées dans l'ordonnancement hors-ligne, la *clairvoyance* et le *déterministe*.

Malgré toutes ces qualités, l'explosion combinatoire reste un grand problème pour que ces méthodes soient utilisables (des résultats empiriques sont montrés dans la section 6.2.4).

Solutions existantes

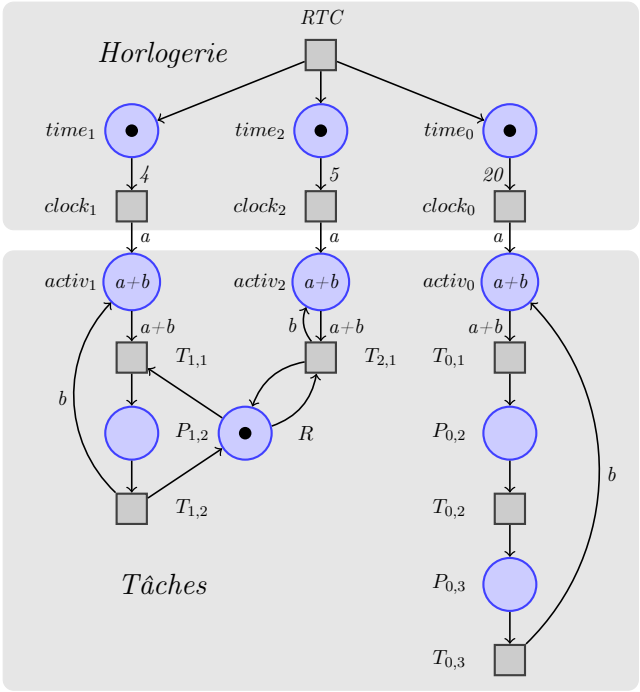
Séparation et évaluation Dans les années 90 plusieurs auteurs se sont intéressés à des méthodes de *séparation et évaluation* [LD60] pour la recherche exhaustive des ordonnancements optimaux. Xu et Parnas [XP90] proposent un premier algorithme pour la recherche d'un ordonnancement pour une architecture monoprocesseur, des tâches sporadiques avec des contraintes de précédence sans préemption. Xu fait une extension de cet algorithme dans [XP93] pour la recherche exhaustive sur une architecture multiprocesseur sans préemption.

En se basant sur ces résultats, Shepard et Gagné [SG91] proposent une recherche, avec les mêmes méthodes, d'un ordonnancement optimal pour des tâches périodiques avec préemption et migration permises. Tous ces algorithmes se basent sur une hypothèse erronée : l'intervalle de faisabilité est $[0, H]$.

Réseaux de Petri Grolleau et Choquet-Geniet [GCG00] proposent une méthode basée sur une modélisation en réseaux de Petri, pour la recherche d'un ordonnancement optimal hors-ligne

pour des ensembles de tâches asynchrones avec des ressources partagées et sur une architecture monoprocesseur. Leur outil PENSMARTS construit une séquence d'exécution à partir de l'étude du graphe de marquage du réseau de Petri. Puisqu'on est dans un cadre monoprocesseur, pour un ensemble synchrone il est suffisant de chercher la séquence jusqu'à l'hyper-période. Dans le cas d'un ensemble asynchrone l'outil s'arrête au plus tard à $\max(O_i) + 2H$.

Exemple 18. *Considérons l'ensemble de tâches* $\mathcal{S} = \{\tau_1 = (0, 4, 4, 2), \tau_2 = (0, 5, 1, 1), \tau_3 = (0, 20, 20, 3)\}$ *et une ressource partagée* R *par* τ_1 *et* τ_2*. Le réseau de Petri représentant le comportement est :*



Dans ce modèle il y a deux parties : la partie supérieure correspond à la description du comportement temporel et la partie inférieure décrit le comportement des tâches temps réel. Une horloge globale RTC (Real Time Clock) "nourrit" de jetons chaque tâche. Pour chaque tâche on dispose d'un accumulateur de jetons, $time_i$*, qui permet de démarrer un nouveau job lorsque le nombre de jetons cumulés est égal à la période.*

Chaque tâche i dispose d'une place, $activ_i$ *qui représente l'initialisation d'un job. Les jetons a et b indiquent qu'on dispose de deux couleurs, a et b. Pour démarrer l'exécution il faut les deux jetons : le jeton a est obtenu à la fin de chaque période ; le jeton b est obtenu à la fin de l'exécution du job précédent. Le temps d'exécution est modélisé par autant de transitions que d'unités de temps à exécuter. Ainsi, la tâche* τ_1 *à 2 transitions* $T_{1,1}$ *et* $T_{1,2}$*.*

Le processeur est une ressource partagée par toutes les tâches et n'est pas représenté ici par souci de lisibilité.

L'analyse des séquences d'ordonnancement se fait à travers les graphes de marquages de réseau de Petri. Lorsqu'un marquage est non valide (dépassement d'échéance), la séquence est avortée et l'algorithme de recherche réalise un retour arrière. Dans le cas de blocage du réseau de Petri, les actions à réaliser sont similaires. Plusieurs critères peuvent être utilisés pour guider la recherche comme maximisation de l'importance de tâches ou minimisation du temps de réponse maximal.

Automates temporisés Dans [BLR05], les auteurs utilisent les automates "priced time" pour chercher un ordonnanceur optimal d'un ensemble de jobs liés par des contraintes de précédence. Ils utilisent le model checker UPPAAL pour réaliser la recherche. Les modèles ne sont pas efficaces et montrent seulement l'applicabilité d'UPPAAL pour la recherche d'ordonnancement hors-ligne.

2.4 Résumé

Nous avons présenté les politiques en-ligne (FP, gEDF, gLLF et LLREF) utilisées dans la suite du manuscrit. Pour ces politiques, il n'existe aujourd'hui aucun travail pour prendre en compte les contraintes de précédence. Nous avons donc proposé un outil d'analyse d'ordonnançabilité réutilisant les méthodes de parcours exhaustif les plus efficaces pour étudier des ensembles réalistes de tâches dépendantes. Nous n'avons pas de problème d'explosion combinatoire pour cette analyse.

Concernant le calcul d'ordonnancement hors-ligne. Il n'existe pas aujourd'hui de travaux pour des ensembles de tâches (dépendantes) multipériodiques asynchrones. Nous allons donc étudier ce problème, même si la taille du problème est très grande et que notre méthode rencontre rapidement des limitations dues à l'explosion combinatoire.

De plus, il n'existe à notre connaissance aucun outillage automatique pour appliquer les méthodes d'analyse de parcours exhaustif. Nous avons donc mis en œuvre un outil prenant en entrée des descriptions simples d'ensembles de tâches et produisant une analyse automatique de leur ordonnancement.

Chapitre 3

Environnement d'exécution sur cible multicœur

Une fois l'ordonnançabilité vérifiée, l'étape suivante est d'exécuter le système sur la cible réelle ou produire une simulation suffisamment réaliste des comportements fonctionnels et temporels. Il faut donc fournir un outil de mise au point au concepteur qui lui permettra d'évaluer le comportement réel de l'application sur la cible. A noter que cet outil doit permettre de choisir différentes politiques d'ordonnancement et également d'observer l'exécution du code fonctionnel utilisateur.

3.1 Objectif

La mise en œuvre d'une chaîne complète de la conception d'un système temps réel multipériodique jusqu'à son exécution sur une cible permet une validation fonctionnelle et temporelle du système. Cela permet entre autre de garantir que des problèmes pratiques n'ont pas été masqués par l'analyse d'ordonnançabilité théorique. L'objectif est d'observer les comportements et de détecter certains problèmes avant l'intégration sur le système réel.

Il y a en particulier 3 aspects à étudier sur les systèmes $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$:

1. *correction fonctionnelle* : une syntaxe correcte est insuffisante pour garantir un fonctionnement correct, il faut également analyser et observer les valeurs des sorties produites à partir des jeux d'entrées. Nous devons prendre en compte, le code fonctionnel utilisateur importé et généré, donc en particulier le protocole de communication en mémoire partagée sous-jacent. Dans le cas de programme PRELUDE, le protocole est celui défini par [For09] qui est généré par le compilateur.
2. *correction temporelle* : il faut vérifier que les tâches s'exécutent en respectant les paramètres temps réel et les contraintes de précédence et ceci pour *différentes politiques d'ordonnancement*. Mixer les corrections fonctionnelle et temporelle revient à observer le comportement théorique d'un programme PRELUDE comme celui illustré dans la figure 1.3 page 7.
3. *les surcoûts liés à l'architecture* : la théorie de l'ordonnancement simplifie les coûts des services exécutifs en les supposant nuls. Or, l'exécution réelle sur cible nécessite l'utilisation :
   - de mécanismes de synchronisation,
   - de changements de contexte ou de migration,
   - de communication en mémoire ou par d'autres moyens.

   En réalité, ces services prennent un certain temps et peuvent donc provoquer des erreurs temporelles. L'objectif d'une exécution fidèle en temps est d'estimer les surcoûts réels de

ces mécanismes. On peut également noter qu'il est théoriquement possible de modéliser ces mécanismes au sein du modèle visant à analyser l'ordonnançabilité de l'ensemble de tâche, par exemple [BWH93]. Il n'y a pas encore à l'heure actuelle de modélisation vraiment fine de ces mécanismes et les résultats sont encore trop pessimistes.

L'objectif de ce chapitre est de passer en revue les solutions existantes permettant de traiter ces différents aspects.

3.2 Simulateurs

Une première famille d'applications permettant d'étudier un système avant l'intégration est celle des *simulateurs*. Un simulateur est une application, qui s'exécute généralement sur une machine différente de la cible, et simule plusieurs types de comportement de l'exécution, comme par exemple, les aspects temporel et fonctionnel. Dans cette section nous présentons plusieurs simulateurs existants permettant d'observer ces (ou partie de ces) comportements.

Simulateurs fonctionnels L'objectif d'un simulateur fonctionnel est de pouvoir évaluer les sorties produites par les fonctions implantées compte tenu de jeux d'entrées. Les langages synchrones disposent en général de simulateur fonctionnel où le temps est représenté de manière logique par des tics. On peut donc exécuter le programme pas à pas (i.e. tic d'horloge par tic d'horloge) et observer les valeurs des entrées et des sorties des nœuds. *Luciole* est un simulateur de ce type dédié à Lustre : il génère des traces fonctionnelles à partir d'un programme Lustre et d'un jeu d'entrées comme la trace montrée dans la figure 1.1 page 5. *Luciole* ne permet pas de simuler un programme Lustre qui ferait appel à des fonctions externes (par exemple écrites en C). On pourrait néanmoins simuler un tel programme en générant le C avec les outils Lustre *poc* ou *ec2c* et en compilant le tout avec le code externe. De la même manière, *Scade Suite* intègre un simulateur fonctionnel pour Scade [Dor08] (version commerciale de Lustre).

Au début de cette thèse, un tel outil n'était pas encore développé pour Prelude. La gestion du temps est un peu plus complexe que l'évaluation des équations à chaque tic puisque Prelude repose sur l'hypothèse synchrone relâchée. Avec Prelude il existe plusieurs traces temporellement valides. Ainsi, produire une trace générique revient à produire des valeurs disponibles dans des intervalles de temps de production et non à des instants. Nous verrons au chapitre 5 que SchedMCore Runner offre suffisamment de fonctionnalités pour permettre une simulation fonctionnelle d'un programme Prelude y compris en prenant en compte les nœuds importés.

Simulateurs d'architecture Il existe également des simulateurs du comportement au niveau instruction des processeurs. On pourrait alors imaginer de coupler la spécification fonctionnelle $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$ et un tel simulateur. Cette approche aurait deux avantages majeurs :

- une observation très fine du comportement. Il serait possible d'étudier l'évolution des instructions, des chargements dans les caches et des conflits sur le bus,
- une évaluation du comportement sur des cibles non disponibles. Le concepteur pourrait également observer le comportement sur des cibles qu'il n'a pas à sa disposition ou qu'il envisage de concevoir.

Il y a également des inconvénients : il faut avoir à sa disposition un tel simulateur (ce qui nécessite d'avoir des contacts privilégiés avec un fondeur) ou être à même de le développer soi-même (ce qui requiert une connaissance fine de la cible et un investissement important de développement).

Les simulateurs d'architecture sont généralement développés en *SystemC*[5] ou *SIMICS*[6]. *SystemC* est un langage pour la description matérielle et logicielle d'un système. Plus précisément, *SystemC* est une extension du langage C++ codée sous forme de bibliothèque de classes composée des composants matériels. Dans la bibliothèque il y a également un standard de moteur de simulation. Plusieurs implantations ont été faites, parmi lesquels TLM-2.0[7] (logiciel officiel du projet *SystemC*) et SystemCASS[8] (outil issu d'un projet de recherche). *SIMICS* est un simulateur de matériel (processeurs, mémoires, dispositifs, bus, ...) sur lequel on peut exécuter, d'une façon similaire à une machine virtuelle, un système d'exploitation avec les pilotes nécessaires. Ainsi, on peut facilement développer une application pour une architecture cible et simuler son comportement sur la cible. Dans le cas d'applications temps réel, ce simulateur permet d'utiliser plusieurs cibles multicœur et d'exécuter des systèmes d'exploitation temps réel. Le fait d'avoir accès au système d'exploitation permettrait de changer l'ordonnanceur et d'implanter les modifications désirées pour la gestion de communications. Pour cette thèse, nous n'avions pas à disposition de simulateur d'une cible réelle et nous ne pouvions donc pas nous tourner vers cette solution.

Graphite[9][MKK+10] est un simulateur d'architectures multicœur un peu différent. L'objectif n'est pas d'être cycle accurate mais de simuler fonctionnellement des systèmes sur une architecture composée d'un nombre très élevé de cœurs (le test proposé dans [MKK+10] concerne une architecture à 1000 cœurs). Graphite est un framework qui s'exécute au dessus du système d'exploitation, pouvant être distribué sur plusieurs machines, et qui permet de simuler une mémoire unique, une affectation transparente pour l'application utilisateur des threads vers les machines, les appels système, les instructions de synchronisation, etc ... L'objectif principal de cette plateforme étant la simulation d'architectures fortement distribuées, l'aspect temporel précis/réaliste n'est pas pris en compte et n'est donc pas assuré. Étant donné le fort contrôle temporel nécessaire à notre modèle, une adaptation de ce simulateur s'avère compliquée.

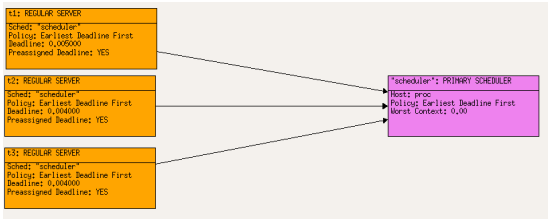
Simulateur de trace d'exécution : MAST[10] est un ensemble d'outils pour la conception des systèmes temps réel développé à l'université de Cantabria. Cette suite d'applications se divise en trois étapes de développement.

La première permet, avec un outil basé sur UML, MAST-UML, de modéliser les aspects matériels et logiciels du système. Dans notre cas, l'architecture matérielle est un processeur défini dans la figure 3.1(b). Il est caractérisé par une horloge (`ticker`) et un ordonnanceur (`scheduler`) de type EDF. Supposons que l'architecture logicielle, décrite à la figure 3.1(a), soit composée de trois tâches périodiques reliées par l'ordonnanceur (`scheduler`). Le comportement périodique est codé comme illustré dans la figure 3.2. Ainsi, chaque tâche est réveillée par l'ordonnanceur toutes les 0.005s.

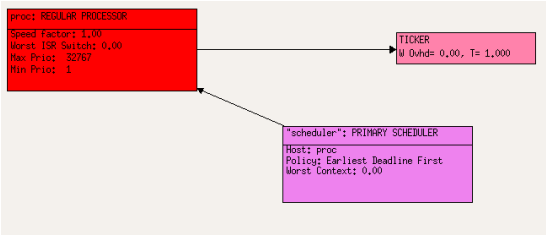
La deuxième étape consiste à analyser le comportement temporel via des techniques d'analyse temporelle comme le calcul de wcet, le calcul automatique de priorités pour une exécution à priorité fixe ou l'analyse de l'ordonnançabilité EDF. L'application dédiée à ce rôle est MAST_analyse qui génère en sortie un fichier XML avec les résultats des analyses demandées.

La troisième étape est la simulation temporelle du système par SIM_MAST. Cette application génère à partir du modèle MAST-UML et du fichier XML de sortie de MAST_analyse, une

5. http://www.systemc.org
6. http://www.windriver.com/products/simics/
7. http://www.systemc.org/downloads/standards/
8. https://www-asim.lip6.fr/trac/systemcass
9. http://groups.csail.mit.edu/carbon/?page_id=111
10. Disponible sur le site http://mast.unican.es



(a) Description de tâches



(b) Définition du processeur et l'ordonnanceur

FIGURE 3.1 – Description de l'architecture

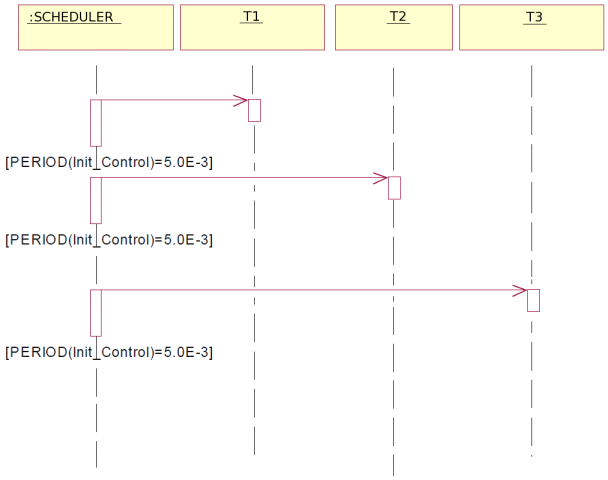


FIGURE 3.2 – Diagramme de séquence des tâches

des possibles traces d'exécution. Elle ne réalise donc qu'une simulation temporelle sans exécuter le code utilisateur.

Simulateur d'ordonnancement multiprocesseur : STORM (Simulation TOol for Real time Multiprocessor scheduling)[UDT10] [11] est un outil de simulation d'ordonnancement pour les systèmes temps réel multiprocesseur. STORM permet de simuler des systèmes à partir de la description de l'ensemble de tâches, de l'architecture matérielle et de la politique d'ordonnancement.

La spécification des architectures et la politique d'ordonnancement se font via un fichier descriptif XML. Le résultat de la simulation est une séquence d'exécution avec deux sorties possibles : soit la visualisation sous forme de chronogramme en STORM, soit l'enregistrement dans des fichiers de sortie pour un traitement postérieur avec des outils extérieurs. La programmation modulaire offre une introduction facile de nouveaux composants, par exemple de nouveaux algorithmes d'ordonnancement. Les contraintes de précédence généralisées ne sont pas prises en compte mais elles pourraient être implantées et utilisées.

Puisque cet outil construit une séquence d'ordonnancement, il pourrait n'être utilisé que partiellement pour l'analyse de l'ordonnançabilité présentée dans la section 2.3. En effet, il ne

11. Disponible sur le site http://storm.rts-software.org/doku.php

peut représenter que les modèles dont on connaît la fenêtre temporelle finie à examiner. De plus, sachant que notre objectif est d'exécuter le code de l'application, une analyse codée en STORM (en Java) n'aurait pas permis de réutilisation entre le code de l'analyse de l'ordonnançabilité et le code d'exécution. En conséquence, il aurait fallu réécrire certaines parties du code et le processus d'analyse/exécution aurait été plus complexe que la solution proposée. Une solution intermédiaire (qui a été testée puis interrompue par manque de temps) était de générer un *modèle* STORM qui pouvait être utile à certaines analyses d'ordonnancement par simulation.

Conclusions sur la simulation Le tableau suivant synthétise les caractéristiques offertes par les simulateurs :

		Simulation		Exécution réelle	Architecture
		Temporelle	Fonctionnelle		
Fonctionnels	Luciole	√	√	√	monoprocesseur
	Scade Suite	√	√	√	monoprocesseur
Architecture	SystemC	√	√	√	multiprocesseur
	SIMICS	√	√	√	multiprocesseur
	Graphite		√	√	multiprocesseur
Trace d'exécution	MAST	√			multiprocesseur
Ordonnancement	STORM	√			multiprocesseur

3.3 Exécution

Aucun des simulateurs existants ne permet d'étudier les aspects fonctionnels lors d'une exécution précise en temps, hormis ceux que nous avons exclus à base de simulateur d'architecture. Une autre possibilité est d'exécuter le code sur une cible multicœur réelle (c'est-à-dire non simulée). L'exécution, combinée avec une technique de traçage permettrait de réaliser des mesures temporelles correctes en plus de valider la correction fonctionnelle. Dans la littérature, on peut trouver plusieurs outils pour l'exécution temps réel. Nous présentons ensuite les outils les plus proches de notre besoin : un *système d'exploitation temps réel* et/ou une *plateforme utilisateur* pour l'exécution.

3.3.1 Système d'exploitation temps réel multiprocesseur

Architecture d'un système avec un RTOS

Une des possibilités pour l'exécution est d'utiliser un système d'exploitation temps réel (*RTOS* pour Real-Time Operating System) fonctionnant sur une architecture matérielle multicœur. Dans la figure 3.3 nous illustrons l'architecture d'un système avec un RTOS. L'architecture cible est représentée en rouge et le RTOS en vert. Les applications à exécuter peuvent être temps réel ou pas.

La mémoire de l'espace noyau du système d'exploitation est généralement séparée de celle de l'espace utilisateur. Certains OS embarqués n'ont pas cette séparation pour des raisons d'efficacité ou de limitation matérielle dans la gestion de l'espace mémoire physique. L'espace noyau est strictement réservé pour l'exécution du noyau et des pilotes de périphérique. En revanche, l'espace utilisateur est l'espace mémoire où les applications utilisateur s'exécutent. Cette séparation permet au système d'exploitation de protéger son espace mémoire et d'éviter qu'une tâche utilisateur ne modifie la mémoire réservée pour l'exécution d'une activité système.

Cette protection mémoire exige que toutes les applications qui nécessitent l'utilisation d'une ressource matérielle, réalisent une demande au système d'exploitation (appel système). Pour faciliter cette communication, le RTOS comprend des pilotes, c'est-à-dire des logiciels ayant des

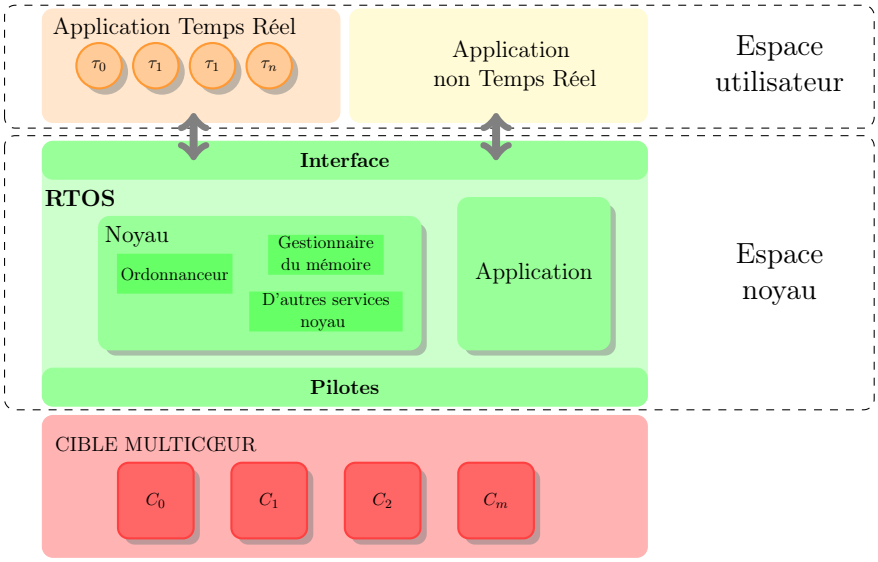


FIGURE 3.3 – Exécution avec un RTOS

droits privilégiés leur permettant d'interagir directement avec les ressources matérielles. De leur coté, les applications utilisateur font appel au système d'exploitation via les primitives qu'il offre pour accéder aux ressources gérées par les pilotes. Les primitives systèmes sont regroupées en *services*, qui définissent les interfaces d'interaction entre les applications et le système. Le noyau dispose de plusieurs services, comme par exemple l'ordonnanceur ou le gestionnaire de mémoire.

Cette solution offre de bonnes performances temporelles : la gestion d'une grande partie des besoins temporels en espace noyau réduit les surcoûts générés à cause des interactions entre le RTOS et les applications ainsi qu'une plus grande priorité des routines noyau qui peut les empêcher d'être interrompues (préemptées). Si l'interface de l'OS ne fournit pas le service que l'application nécessite, alors il faut éventuellement changer le noyau du RTOS pour l'ajouter. En revanche, une modification de ce type implique des connaissances précises du noyau et le cycle de développement est long et délicat, car une nouvelle modification requiert une reconstruction et mise en place du noyau.

Classification des RTOS

Un grand nombre de RTOS existe dans la littérature. Un premier groupe englobe les RTOS commerciaux comme VxWorks [12], LynxOS [13] ou PikeOS [14]. Nous avons néanmoins choisi de n'utiliser que des solutions libres afin de partager le code avec la communauté.

Patch Linux Parmi les solutions libres existantes, une famille est composée des solutions implantées en partant d'un système Linux. Bien que le noyau Linux offre un grand nombre de services pour maîtriser le temps d'exécution des tâches, son exécution par défaut ne suffit pas (encore ?) à une exécution temps réel dure. C'est pour cela que les solutions basées sur Linux ajoutent des services noyau temps réel grâce à l'introduction d'un patch (modification du noyau

12. http://www.windriver.com/products/vxworks/
13. http://www.lynuxworks.com/rtos/rtos.php
14. http://www.sysgo.com/products/pikeos-rtos-technology/

qu'on peut inclure au moment de la compilation). Comme exemple de ces systèmes on peut citer : Litmus^{RT} [15], RESCH [16], Linux SCHED_DEADLINE [17], Linux/RK [18] ou Xenomai [19].

Une solution à base de patch Linux a l'avantage de s'exécuter dans l'espace noyau avec une bonne performance et une limitation des surcoûts. Par contre, un patch est codé pour une version précise du noyau et donc chaque évolution du noyau Linux peut nécessiter la mise à jour de ce patch. Vue la vitesse d'évolution du noyau (une version toutes les 9 semaines environ), l'évolution du patch devrait se faire à la même vitesse, ce qui demande un effort de développement conséquent.

Une autre difficulté pour l'intégration de nos politiques d'ordonnancement dans un RTOS (Linux compris) est la limitation du système pour exprimer les paramètres temps réel d'une tâche. Dans la plupart des systèmes, on ne dispose que de la priorité système de la tâche [20]. Les précédences constituent pour nous un paramètre d'ordonnancement comme un autre utilisé par l'ordonnanceur au moment de l'exécution. C'est la raison pour laquelle il nous paraît primordial de pouvoir adapter la politique d'ordonnancement, si le système ne prévoit pas de base les paramètres nécessaires.

Présentons plus en détail la solution Litmus^{RT} (LInux Testbed for MUltiprocessor Scheduling in Real-Time systems). Le projet Litmus^{RT} [CLB+06] est une extension temps réel du noyau Linux (la dernière version Litmus^{RT} 2011.1 utilise le noyau Linux 2.6.36) dédiée à l'ordonnancement et la synchronisation temps réel sur des architectures multiprocesseur. Le noyau a été modifié pour supporter un ensemble de tâches sporadiques et des ordonnanceurs modulaires. Quatre ordonnanceurs sont actuellement disponibles : EDF partitionné (P-EDF), gEDF (gEDF), clustered EDF (C-EDF) et PFAIR. L'objectif de Litmus^{RT} est de fournir une plateforme expérimentale pour la recherche en systèmes temps réel. Pour cela, Litmus^{RT} met à disposition des utilisateurs des abstractions et des interfaces du noyau Linux afin de simplifier le prototypage des ordonnanceurs temps réel multiprocesseurs et les algorithmes de synchronisation. Un autre objectif est de montrer que des algorithmes d'ordonnançabilité complètement dynamiques, comme PFAIR, peuvent être implantés sur les architectures actuelles, même si l'industrie y est réticente. De plus, Litmus^{RT} offre également trois applications pour l'analyse de traces qui permettent d'analyser l'ordonnancement et les surcoûts :

- *litmus_log* : les fichiers log de Litmus^{RT} permettent de récupérer des informations utiles pour le debug. Ce mécanisme peut être désactivé car ce type de traçage ajoute lui-même des surcoûts.
- *ft_trace* : cet outil est utilisé pour le traçage des surcoûts du système. Il est alors possible de minimiser la génération des surcoûts et leur impact temporel.
- *sched_trace* : un outil qui permet d'enregistrer les évènements du système, tel que la fin ou le réveil d'un job. Cette application stocke ces informations dans un buffer différent pour chaque processeur. *sched_trace* étant basé sur *ft_trace*, ne génère que peu de surcoûts.

Systèmes indépendants En plus des systèmes basés sur Linux, il existe une multitude de RTOS open source autonomes. En multiprocesseur on peut trouver, notamment, RTEMS [21],

15. http://www.cs.unc.edu/~anderson/litmus-rt/
16. http://users.soe.ucsc.edu/~shinpei/resch/
17. http://www.evidence.eu.com/content/view/313/390/
18. http://www.cs.cmu.edu/~rajkumar/linux-rk.html
19. http://www.xenomai.org
20. des efforts pour intégrer la notion de deadline sont en cours http://www.evidence.eu.com/content/view/313/390/ [FCTS09]
21. http://www.rtems.org/

eCos [22] (limité à 8 cœurs) et ERIKA [23] (seulement ordonnanceur partitionné). D'autres systèmes existent également pour les systèmes monoprocesseurs, par exemple, MaRTE OS [24], FreeRTOS [25] ou Trampoline [BBFT06] [26].

Bien que conçus pour des systèmes temps réel, contrairement aux solutions Linux+Patch, chaque solution dispose d'une représentation du modèle de tâche propre, en général, assez éloigné du nôtre. Ainsi, aucun ordonnanceur proposé n'est capable de gérer notre modèle de précédences. La modification de la politique d'ordonnancement génère les mêmes problèmes que ceux exposés dans le cas d'un patch Linux.

Expliquons plus en détail MaRTE OS. MaRTE OS [RH02] est un système d'exploitation temps réel dur pour des applications embarquées respectant l'ensemble Real-Time POSIX.13. MaRTE OS est une implantations d'un système d'exploitation autonome et ne dépend donc d'aucun autre système. L'approche choisie dans le noyau MaRTE OS pour l'ordonnancement de tâches est hiérarchique. Ainsi, le noyau dispose d'un ordonnanceur à priorité fixe implanté en espace noyau qui exécute deux types de tâches : (1) *Tâches d'ordonnancement* : ces tâches ont une priorité plus élevée et leur fonction est l'implantation d'une politique d'ordonnancement ; (2) *Tâches d'application* sont les tâches de l'application à exécuter. Cette séparation entre l'ordonnanceur système et la politique d'ordonnancement permet aisément d'ajouter une nouvelle politique. J. Forget dans sa thèse [For09] adapte la politique EDF implantée dans le système pour la gestion des contraintes de précédence dans le cas monoprocesseur. MaRTE OS dispose de deux modes d'exécution : mode simulateur qui peut s'exécuter sur un système Linux de façon à simuler l'exécution de l'application sans garantie temporelle ; mode exécution réel disponible uniquement pour des architectures x86. Cette dichotomie permet de tester facilement les applications sur les machines de développement puis d'implanter le système sur la machine désirée une fois que la simulation est correcte.

Ce système d'exploitation est le plus proche de nos besoins mais malheureusement il est uniquement disponible pour des systèmes monoprocesseur. Malgré tout, certains des concepts nous seront utiles dans la solution proposée par cette thèse.

Résumé des RTOS Dans le tableau suivant nous montrons une synthèse des RTOS cités (Linux supporte les architectures multicœur d'où la case grise) :

		Basé sur Linux	**Non basé sur Linux**
Multiprocesseur	**Commercial**		PikeOs LynxOS VxWorks
	Non Commercial	LitmusRT Xenomai Linux SCHED_DEADLINE LINUX/RK RESCH	ERIKA eCos RTEMS
Monoprocesseur	**Commercial**		N/A
	Non Commercial		MarteOS FreeRTOS Trampoline

Les RTOS ne sont généralement pas spécifiés pour pouvoir modifier la politique d'ordonnancement (des exceptions existent comme MaRTE OS).

22. http://ecos.sourceware.org/docs.html
23. http://erika.tuxfamily.org/
24. http://marte.unican.es/
25. http://interactive.freertos.org/entries/20423292-freertos-multicore
26. http://trampoline.rts-software.org/

3.3.2 Solution intermédiaire : Meta-scheduler

Le Meta-scheduler [LRSF04] est une plateforme, développée à l'université de Virginia Tech, servant à l'exécution des applications temps réel. Cette solution est une sur-couche qui agit entre le noyau du système d'exploitation et l'application utilisateur pour fournir les services temps réel nécessaires.

Architecture du Meta-scheduler

La figure 3.4 décrit l'architecture du Meta-scheduler. La plateforme s'exécute au-dessus d'un système d'exploitation (non nécessairement temps réel). D'autres applications non temps réel peuvent s'exécuter sur le système en parallèle.

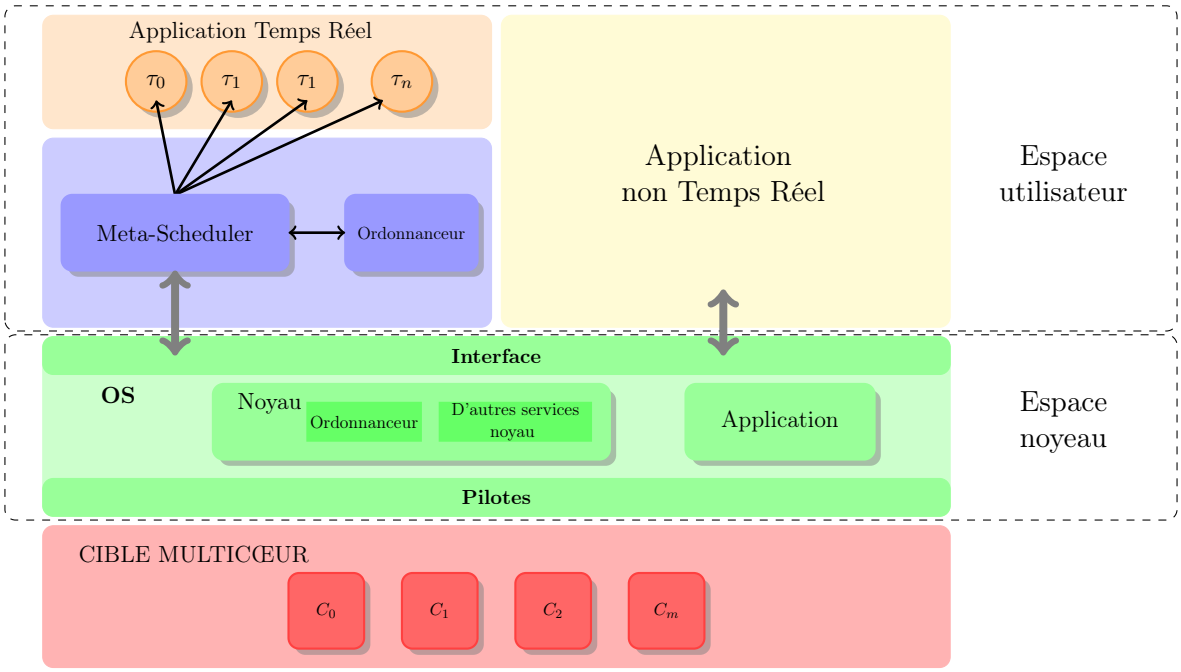


FIGURE 3.4 – Exécution avec le Meta-scheduler

La mission du Meta-scheduler est alors de produire une gestion temps réel sur un système non temps réel. Il doit d'une part implanter une politique d'ordonnancement temps réel pour la fournir aux tâches temps réel et d'autre part interagir avec le système d'exploitation à travers l'interface (méthodes de synchronisation, horloges,...) pour gérer le temps et l'exécution des tâches. Il faut remarquer que ce nouvel ordonnancement se réalise au-dessus de l'ordonnancement du système d'exploitation. Il est alors nécessaire, pour produire un ordonnancement correct, que l'ordonnanceur du système d'exploitation soit préemptible à tout moment, pour pouvoir exécuter les tâches selon les contraintes temporelles.

Le Meta-scheduler propose une conception modulaire des politiques d'ordonnancement avec une séparation entre une librairie générique, les tâches et la fonction d'ordonnancement elle-même. Dès lors, la mise en place d'une nouvelle politique d'ordonnancement se simplifie et consiste à coder une nouvelle fonction qui sera liée à librairie du Meta-scheduler et aux fonctions à exécuter au moment de la compilation.
L'approche d'une solution en espace utilisateur présente plusieurs avantages :

- *portabilité* : le Meta-scheduler respecte/utilise la norme POSIX pour la maîtrise du temps et l'exécution des tâches. Par conséquent, tout système d'exploitation respectant cette norme peut exécuter le Meta-scheduler ;
- *facilité* : l'application s'exécute en espace utilisateur et est indépendante des modifications du noyau.

Cette solution permet d'exécuter facilement une application ainsi que de modifier la politique d'ordonnancement désirée. En revanche, l'implantation est uniquement monoprocesseur et actuellement le développement de ce code semble être abandonné en n'étant plus public sur le site du groupe développeur. De plus, les interactions entre le système d'exploitation, le Meta-scheduler et les tâches temps réel, ainsi que les interruptions produites par les applications non temps réel qui s'exécutent en parallèle, peuvent produire des surcoûts difficilement mesurables.

3.4 Résumé

Dans cette section nous avons présenté différentes options pour simuler et exécuter un ensemble de tâches temps réel conforme à notre modèle de tâche. Les simulateurs temporels et fonctionnels permettent de vérifier la correction fonctionnelle et temporelle. Les systèmes d'exploitation temps réel permettent d'exécuter le code. Malheureusement, les solutions existantes ne répondent pas complètement aux besoins de cette thèse. En premier lieu, les solutions commerciales sont hors sujet du cadre de cette thèse. Les solutions académiques libres sont pour la plupart conçues comme un patch du Linux standard. Ainsi, une exécution basée sur un système d'exploitation dépendrait excessivement de son évolution (une modification du noyau peut rendre un patch incompatible). Les solutions les plus adaptées à nos besoins sont MARTE OS et le Meta-scheduler, disponibles uniquement sur architectures monocœur.

Nous proposons donc un nouveau framework basé sur les idées de MARTE OS, comme la conception modulaire qui nous permet de greffer aisément un nouvel ordonnanceur, et du Meta-scheduler comme l'idée de plateforme d'exécution en mode utilisateur au dessus d'un système d'exploitation offrant les services nécessaires. La conception et la réalisation de cet outil seront présentées au chapitre 5.

Deuxième partie

Contributions

Chapitre 4

Modélisation d'un système temps réel

Ce chapitre débute la description du travail réalisé au cours de cette thèse. Notre objectif est de fournir un environnement de développement de systèmes multipériodiques programmés en PRELUDE. Nous avons expliqué dans l'introduction figure 3 page xiii que la génération de code repose sur deux axes : le premier concerne le choix de stratégies correctes d'ordonnancement et le deuxième concerne l'exécution du code généré. Cette partie est dédiée à la question de l'ordonnancement multiprocesseur. L'ensemble de tâches généré à partir d'un programme PRELUDE est de la forme $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$ et l'objectif de l'ordonnancement est d'appliquer une politique valide sur $(\mathcal{S}, \mathcal{R})$ afin de donner temporellement l'accès aux processeurs de façon à respecter les contraintes temps réel et de précédence. Nous avons choisi une approche à base de parcours exhaustif d'une part car aucune solution analytique actuelle ne prend en compte les précédences et d'autre part car les performances obtenues par notre outil (comme nous le verrons ultérieurement chapitre 6) sont suffisamment bonnes pour traiter des ensembles de tâches réalistes.

Dans ce chapitre on établit les modèles formels représentant les comportements temporels des tâches $\mathcal{S}$ et des contraintes de précédence $\mathcal{R}$. Nous avons outillé l'ensemble de nos contributions dans l'environnement SCHEDMCORE et tout ce qui concerne la question de l'ordonnancement se trouve dans l'outil SCHEDMCORE CONVERTER. L'idée générale est de traduire l'ensemble de tâches et la question en cours de traitement (vérification de la correction d'un ordonnancement, recherche d'une affectation de priorité fixe valide ou recherche d'un ordonnancement hors-ligne) en des programmes équivalents C ou UPPAAL. Nous montrons donc dans ce chapitre partie 4.1 les codages équivalents pris en paramètres des modèles UPPAAL et C. A noter qu'UPPAAL prend entrée un sous-ensemble du langage C pour les déclarations et les fonctions à appliquer sur les transitions, ainsi dans ce chapitre, il n'y aura que des déclarations en C prises en compte à la fois en UPPAAL et C. Dans la partie 4.2, nous détaillons les techniques de parcours exhaustif utilisées pour l'analyse d'ordonnançabilité des politiques en-ligne. Enfin, la partie 4.3 est dédiée à la génération de paramètres hors-ligne : affectation de priorités valides pour FP et ordonnancement hors-ligne.

4.1 Encodage de l'évolution des configurations des tâches

L'ensemble des travaux présentés dans la section 2.3.2 repose sur une représentation du comportement des tâches par des modèles formels. Les versions synthétiques de Cucu et Goossens [CG06], Baker et Cirinei [BB07] et Guan et al. [GGL+08] utilisent la notion de configuration à l'instant t comme le restant des paramètres temps réel de chaque tâche jusqu'à la prochaine période. Tous les travaux utilisent également des pas de temps discret afin de mettre à jour les

configurations. Nous reprenons ces concepts et nous les généralisons pour prendre en compte tous les paramètres temps réel ainsi que les précédences.

4.1.1 Encodage des configurations des tâches

Une configuration à un instant t décrit l'état d'une tâche à cet instant. D'un point de vue temporel, il s'agit du temps restant des paramètres jusqu'au réveil du prochain job. Plus formellement,

Définition 6 (Configuration d'une tâche)**.** *Toute tâche τ_i à un instant t est caractérisée par le tuple*

$$conf(\tau_i, t) = (O_i(t), T_i(t), D_i(t), C_i(t))$$

avec

- $O_i(t) : \mathbb{N} \to [0, O_i]$ *est une fonction qui à un instant t représente le temps restant jusqu'à l'activation de la première instance. Si la tâche est inactive, c'est-à-dire si elle n'a pas commencé son exécution, la valeur de $O_i(t)$ est strictement positive :*

$$O_i(t) = \max(O_i - t, 0)$$

- $T_i(t) : \mathbb{N} \to [0, T_i]$ *est une fonction qui à un instant t représente le temps restant jusqu'au réveil du prochain job. Une fois un job activé, la valeur de la période décroît progressivement de T_i à 1.*

$$\begin{cases} T_i(t) = T_i - ((t - O_i) mod\ T_i) & \text{si } O_i(t) = 0 \\ T_i(t) = T_i & \text{si } O_i(t) > 0 \end{cases}$$

- $D_i(t) : \mathbb{N} \to [0, D_i]$ *est une fonction qui à un instant t représente le temps restant jusqu'à l'échéance du job courant. L'échéance décroît aussi progressivement de D_i à 0 quand le job est actif. Cependant, comme le modèle est à échéance contrainte, $D_i(t)$ peut être inférieur à $T_i(t)$. Alors, pendant le temps entre la fin de l'échéance et l'initialisation d'un nouveau job, la valeur de $D_i(t) = 0$.*

$$\begin{cases} D_i(t) = \max(0, T_i(t) - (T_i - D_i)) & \text{si } O_i(t) = 0 \\ D_i(t) = D_i & \text{si } O_i(t) > 0 \end{cases}$$

- $C_i(t) : \mathbb{N} \to [0, C_i]$ *est une fonction qui à un instant t représente le temps d'exécution restant du job en cours d'exécution. La mise à jour se fait en deux étapes : il faut d'abord calculer le temps restant d'exécution en fonction de l'accès au processeur puis remettre à jour si un nouveau job se réveille. La première mise à jour dépend de ce qui a déjà été exécuté selon les choix de la politique d'ordonnancement. On note $P(t)$ l'ensemble des tâches les plus prioritaires qui accèdent à un processeur à l'instant t.*

$$\begin{cases} C_i(t) = 0 & \text{si } O_i(t) > 0 \text{ ou si } t = 0 \\ C_i(t) = C_i(t-1) - 1 & \text{si } i \in P(t-1) \text{ et } O_i(t-1) = 0 \text{ et } t > 0 \\ C_i(t) = C_i(t-1) & \text{sinon} \end{cases}$$

Puis la deuxième mise à jour regarde si un nouveau job se réveille :

$$\begin{cases} C_i(t) = C_i & \text{si } ((t - O_i) mod\ T_i = 0 \text{ et } C_i(t) = 0 \text{ et } O_i(t) = 0) \text{ ou si } O_i(t) > 0 \\ C_i(t) = C_i(t) & \text{sinon} \end{cases}$$

L'état du système à l'instant t est l'ensemble des configurations des tâches à cet instant.

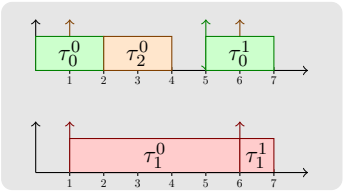
Définition 7 (État du système)**.** *Pour tout ensemble de tâches $\mathcal{S} = \{\tau_i\}_{i=1,..,n}$ nous définissons l'état du système conf(t) à un instant t comme conf(t) $= \langle conf(\tau_1, t), \ldots, conf(\tau_n, t)\rangle$.*

Une exécution est alors une succession de configurations.

Définition 8 (Sequence d'exécution)**.** *Une séquence d'exécution sur l'intervalle $[a, b]$ (avec $a < b$) est définie comme l'ensemble des configurations $s(a, b) = conf(a), conf(a + 1), \cdots, conf(b)$.*

Exemple 19. *Nous désirons exécuter l'ensemble de tâches $\{\tau_0 = (0, 5, 5, 2), \tau_1 = (1, 5, 5, 5), \tau_2 = (1, 5, 5, 2)\}$ sur une architecture à deux processeurs selon une politique à priorité fixe avec $\tau_0 < \tau_1 < \tau_2$.*

time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
τ_0	$(0, 5, 5, 2)$	$(0, 4, 4, 1)$	$(0, 3, 3, 0)$	$(0, 2, 2, 0)$	$(1, 1, 0, 0)$	$(0, 5, 5, 0) \to (0, 5, 5, 2)$	$(0, 4, 4, 1)$	...
τ_1	$(1, 5, 5, 5)$	$(0, 5, 5, 5)$	$(0, 4, 4, 4)$	$(0, 3, 3, 3)$	$(0, 2, 2, 2)$	$(0, 1, 1, 1)$	$(0, 5, 5, 0) \to (0, 5, 5, 5)$	...
τ_2	$(1, 5, 5, 2)$	$(0, 5, 5, 2)$	$(0, 4, 4, 2)$	$(0, 3, 3, 1)$	$(0, 2, 2, 0)$	$(0, 1, 1, 0)$	$(0, 5, 5, 0) \to (0, 5, 5, 2)$	...
P(t)	$\{0\}$	$\{0, 1\}$	$\{1, 2\}$	$\{1, 2\}$	$\{1\}$	$\{0, 1\}$	$\{0, 1\}$	...



A l'instant initial, la configuration de chaque tâche correspond aux paramètres temps réel $\forall i = 0..2$, $T_i(0) = T_i$, $D_i(0) = D_i$, $O_i(0) = O_i$ et $C_i(0) = C_i$. τ_0^0 peut s'exécuter car sa date de réveil est nulle. En revanche, τ_1^0 et τ_2^0 ne peuvent s'exécuter car leur date de démarrage est 1. A l'instant 1, les paramètres de tous les jobs sont actualisés, ainsi pour τ_0 on a $T_0(1) = T_0(0) - 1 = 4$, $D_0 = D_0(0) - 1 = 4$, $C_0 = C_0(0) - 1 = 1$. De leur côté, τ_1^0 et τ_2^0 se réveillent. Comme les trois tâches sont actives, on regarde l'ensemble $P(1)$ qui indique que les deux tâches les plus prioritaires sont τ_0 et τ_1. A l'instant 2, τ_0^0 termine son exécution libérant les deux processeurs pour les deux jobs actifs jusqu'à 5 où le job τ_0^1 se réveille. A cet instant, on montre dans le tableau la mise à jour en 2 étapes de $C_0(t)$ pour τ_0. L'ordonnancement est valide si la valeur intermédiaire de $C_0(t)$ est nulle.

Regardons plus précisément la façon dont serait codée cette exécution par les modèles des auteurs présentés dans l'état de l'art :

- *selon le modèle de Cucu et Goossens [CG06] :*

time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
τ_0	$(1, 0, 2)$	$(1, 1, 1)$	$(1, 2, 0)$	$(1, 3, 0)$	$(1, 4, 0)$	$(1, 5, 0) \to (1, 0, 2)$	$(1, 1, 1)$	...
τ_1	$(-1, 1, 0)$	$(1, 0, 5)$	$(1, 1, 4)$	$(1, 2, 3)$	$(1, 3, 2)$	$(1, 4, 1)$	$(1, 5, 0) \to (1, 0, 5)$	...
τ_2	$(-1, 1, 0)$	$(1, 0, 2)$	$(1, 1, 2)$	$(1, 2, 1)$	$(1, 3, 0)$	$(1, 4, 0)$	$(1, 5, 0) \to (1, 0, 2)$	...

- *selon le modèle de Baker et Cirinei [BB07], l'avancement temporel correspond à une transition* clock-tick*. Les flèches indiquant l'activation d'un job correspondent aux transitions du type* ready *:*

time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
τ_0	$(0, 0) \to (5, 2)$	$(4, 1)$	$(3, 0)$	$(2, 0)$	$(1, 0)$	$(0, 0) \to (5, 2)$	$(4, 1)$	...
τ_1	$(0, 0)$	$(0, 0) \to (5, 5)$	$(4, 4)$	$(3, 3)$	$(2, 2)$	$(1, 1)$	$(0, 0) \to (5, 5)$	...
τ_2	$(0, 0)$	$(0, 0) \to (5, 2)$	$(4, 2)$	$(3, 1)$	$(2, 0)$	$(1, 0)$	$(0, 0) \to (5, 2)$	...

– *selon le modèle de Guan et al. [GGL+08] (ne permet pas de représenter les tâches asynchrones) :*

time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
τ_0	$(2,5)$	$(1,4)$	$(0,3)$	$(0,2)$	$(0,1)$	$(0,0) \rightarrow (2,5)$	$(1,4)$	...
τ_1	*non représentable*	$(5,5)$	$(4,4)$	$(3,3)$	$(2,2)$	$(1,1)$	$(0,0) \rightarrow (5,5)$	...
τ_2	*non représentable*	$(2,5)$	$(2,4)$	$(1,3)$	$(0,2)$	$(0,1)$	$(0,0) \rightarrow (2,5)$	...

Codage en C des configurations Pour homogénéiser les solutions, l'encodage en C proposé est identique pour les modèles C et UPPAAL. Comme UPPAAL n'implémente pas tout le standard ANSI C mais seulement un sous ensemble, plusieurs mécanismes C ne peuvent être utilisés comme la mémoire dynamique ou les structures.

La configuration des tâches est stockée dans plusieurs vecteurs de taille statique égale au nombre de tâches n. Pour chaque attribut de tâche (T_i, D_i, C_i) on utilise deux listes, la première a des valeurs variables pour stocker l'évolution du système, `int list_attr_var[n]` ; la deuxième est constante et contient les attributs initiaux `const int list_attr[n]`. Pour la date de réveil, on utilise une liste unique `int list_o[n]`. Alors, le tuple `(list_o[i], list_c_var[i], list_d_var[i], list_t_var[i])` est équivalent à $conf(\tau_i, t)$ tandis que `(list_o[i], list_c[i], list_d[i], list_t[i])` est égal à (O_i, C_i, D_i, T_i).

Exemple 20. *Reprenons l'exemple de l'ensemble de tâches précédent* $\{\tau_0 = (0,5,5,2), \tau_1 = (1,5,5,5),\ \tau_2 = (1,5,5,2)\}$. *La déclaration en C est alors :*

Algorithme 4.1: Déclaration des paramètres des tâches

```
1  typedef int[0,1] taskOffsetRange;
2  typedef int[0,5] taskDeadlineRange;
3  typedef int[0,5] taskWCETRange;
4  typedef int[0,5] taskPeriodRange;

5  const int TASK_NUMBER=3;
6  const taskWCETRange list_c_var[TASK_NUMBER] = {2 , 5 , 2};
7  const taskDeadlineRange list_d_var[TASK_NUMBER]= {5 , 5 , 5};
8  const taskPeriodRange list_t_var[TASK_NUMBER] = {5 , 5 , 5};
9  taskOffsetRange list_o[TASK_NUMBER] = {0 , 1 , 1};
10 taskWCETRange list_c[TASK_NUMBER] = {2 , 5 , 2};
11 taskDeadlineRange list_d[TASK_NUMBER]= {5 , 5 , 5};
12 taskPeriodRange list_t[TASK_NUMBER] = {5 , 5 , 5};
13 const int H =5;
```

Les quatre premières lignes correspondent à la déclaration de types bornés pour chacun des paramètres. La borne supérieure de chaque attribut est le maximum de cet attribut entre toutes les tâches. Ensuite, on trouve la déclaration du nombre de tâche, les déclarations des vecteurs constants et des vecteurs variables. Ils sont bien évidemment identiques à l'instant 0. *Enfin, on déclare l'hyper-période* $H = ppcm(T_i) = 5$.

La fonction décrivant l'évolution des configurations, montrée dans l'algorithme 4.2 , et actualisant les paramètres des tâches est la fonction `step()`. Pour chaque tâche, on réduit de 1 les valeurs de T_i, D_i de toutes les tâches actives. Si la tâche i est prioritaire, elle accède à un processeur et sa durée d'exécution est décrémentée de 1. Nous verrons ultérieurement comment la condition $prio(i)$ est codée. La ligne 9 réinitialise les valeurs de (T_i, D_i, C_i) de toutes les tâches dont un nouveau job se réveille et qui ont correctement achevé le job précédent. En la ligne 16 réduit de 1 la date d'activation des tâches encore inactives $(O_i > 0)$.

Optimisation de l'encodage L'encodage précédent est assez verbeux. En effet, il suffit de connaître $C_i(t)$ car les autres paramètres peuvent facilement être recalculés à partir de la connaissance de l'instant courant t.

Fonction 4.2 : step : Fonction d'évolution temporelle

```
1  pour i ← 0 to TASK_NUMBER faire
2      si list_o[i]=0 alors
3          si list_t_var[i] > 0 alors
4              list_t_var[i]–;
5          si list_d_var[i] > 0 alors
6              list_d_var[i]–;
7          si prio(i) alors
8              list_c_var[i]–;
9          si list_t_var[i]=0 alors
10             si list_c_var[i]=0 alors
11                 list_t_var[i]=list_t[i];
12                 list_w_var[i]=list_w[i];
13                 list_d_var[i]=list_d[i];
14     sinon
15         list_o[i]–;
```

Définition 9 (Configuration optimisée d'une tâche)**.** *Toute tâche τ_i à un instant t est caractérisée simplement par $C_i(t)$. $C_i(t)$ se calcule également en deux étapes, calcul du temps restant d'exécution en fonction de l'accès au processeur :*

$$\begin{cases} C_i(t) = 0 & \text{si } O_i \geq t \\ C_i(t) = C_i(t-1) - 1 & \text{si } i \in P(t-1) \\ C_i(t) = C_i(t-1) & \text{sinon} \end{cases}$$

Puis la deuxième mise à jour regarde si un nouveau job se réveille :

$$\begin{cases} C_i(t) = C_i & \text{si } ((t - O_i) mod\ T_i = 0 \text{ et } C_i(t) = 0) \text{ ou si } O_i \geq t \\ C_i(t) = C_i(t) & \text{sinon} \end{cases}$$

Exemple 21. *Reprenons l'exemple 19 et montrons l'évolution temporelle sur le modèle optimisé :*

time	*0*	*1*	*2*	*3*	*4*	*5*	*6*	...
τ_0	2	1	0	0	0	$0 \to 2$	1	...
τ_1	5	5	4	3	2	1	$0 \to 5$	...
τ_2	2	2	2	1	0	0	$0 \to 2$	...
P(t)	$\{0\}$	$\{0,1\}$	$\{1,2\}$	$\{1,2\}$	$\{1\}$	$\{0,1\}$	$\{0,1\}$	...

Codage en C des configurations optimisées Pour stocker la description statique de la tâche, on utilise toujours les 4 vecteurs qui contiennent les attributs initiaux, `const int list_attr[n]`. Alors, `(list_o[i], list_c[i], list_d[i], list_t[i])` est égal à (O_i, C_i, D_i, T_i). On conserve le vecteur `list_c_var[i]` pour $C_i(t)$. On utilise deux entiers `time`$\in [0, H-1]$ et `deb`$\in [0, \max(O_i)-1]$ pour stocker l'instant courant.

Exemple 22. *Les déclarations* C *de la configuration optimisée pour l'ensemble* $\{\tau_0 = (0,5,5,2), \tau_1 = (1,5,5,5), \tau_2 = (1,5,5,2)\}$ *sont donnés dans 4.3.*

La fonction `step()`, illustrée dans l'algorithme 4.4, est légèrement modifiée. Elle est décomposée en deux étapes : la première étape est la mise à jour de l'instant courant, l'entier `deb` est incrémenté jusqu'à la date de réveil la plus grande `MaxO`, puis l'entier `time` est incrémenté modulo l'hyper-période H. La deuxième boucle, ligne 7, décrémente la valeur du temps d'exécution si la

Algorithme 4.3: Déclaration des paramètres optimisés

```
1  typedef int[0,1] taskOffsetRange;
2  typedef int[0,5] taskDeadlineRange;
3  typedef int[0,5] taskWCETRange;
4  typedef int[0,5] taskPeriodRange;

5  const int TASK_NUMBER=3;
6  taskWCETRange list_c_var[TASK_NUMBER] = {2 , 5 , 2};
7  const taskWCETRange list_c[TASK_NUMBER] = {2 , 5 , 2};
8  const taskDeadlineRange list_d[TASK_NUMBER]= {5 , 5 , 5};
9  const taskPeriodRange list_t[TASK_NUMBER] = {5 , 5 , 5};
10 const taskOffsetRange list_o[TASK_NUMBER] = {0 , 1 , 1};
11 const int H =5;
12 const int MaxO =1;
13 int time=0;
14 int deb=0;
```

tâche est prioritaire, c'est-à-dire si elle se trouve dans `P(t)`, puis réinitialise le WCET lorsqu'un nouveau job se réveille.

Fonction 4.4 : stepOptimise

```
1  si deb < MaxO alors
2      deb++;
3  sinon
4      time = (time+1)% H;
5  pour i ← 0 to TASK_NUMBER faire
6      si prio(i) alors
7          list_c_var[i]–;
8      si deb ≥ list_o[i] alors
9          si (time+deb-list_o[i]) mod list_t[i]=0 alors
10             si list_c_var[i]=0 alors
11                 list_c_var[i]=list_c[i];
```

4.1.2 Représentation des contraintes de précédence

Une contrainte de précédence entre deux tâches $\tau_i \xrightarrow{M_{i,j},L} \tau_j$ est composée d'un ensemble de mots $M_{i,j} = \{w_0, \ldots, w_k\}$ qui s'appliquent sur une longueur L. On note $N_{i,j} = \#M_{i,j}$ le nombre de mots de l'ensemble des contraintes de précédence. Chaque mot est composé de deux paramètres (m, m') tels que $\tau_i^m \rightarrow \tau_j^{m'}$, $m \in [0, (ppcm(T_i, T_j) \times L)/T_i - 1]$ et $m' \in [0, (ppcm(T_i, T_j) \times L)/T_j - 1]$.

Définition 10 (Contrainte de précédence à un instant t)**.** *On peut définir l'état de la contrainte de précédence de $\tau_i \xrightarrow{M_{i,j},L} \tau_j$ à l'instant t comme l'ensemble des mots actifs $M_{i,j}(t) = \{(m_1, m'_1), \ldots, (m_{k_t}, m'_{k_t})\}$. Puisque nous raisonnons en relatif, $M_{i,j}(t) \subseteq M_{i,j}$. On a $M_{i,j}(0) = M_{i,j}$.*

Soit la fonction $e_{\tau_i} : \mathbb{N} \rightarrow \mathbb{N}$ où $e_{\tau_i}(k)$ est l'instant de finalisation du job τ_i^k. Cela revient à dire que $C_i(t) = 0$, $C_i(t-1) = 1$ et $t = e_{\tau_i}(k)$. L'évolution des précédences dépend alors de deux événements :

- une contrainte de précédence $(m, m') \in M_{i,j}(t-1)$ est relâchée à l'instant t si t correspond à la fin de l'exécution de τ_i^n, c'est-à-dire si $t = e_{\tau_i}(n)$ avec $n = m + k\frac{pL}{T_i}$, $k \in \mathbb{N}$ et

$p = ppcm(T_i, T_j)$. La mise à jour des contraintes actives se fait de la façon suivante :

$$\begin{cases} M_{i,j}(t) = M_{i,j}(t-1) \setminus \{(m, m')\} & \text{si } t = e_{\tau_i}(m + k\frac{pL}{T_i}) \\ M_{i,j}(t) = M_{i,j}(t-1) & \text{sinon} \end{cases}$$

– une contrainte de précédence (m, m') s'active à t lorsque une nouvelle hyper-période $ppcm(T_i, T_j)$ démarre, c'est-à-dire lorsque $t = O_j + k \times L \times ppcm(T_i, T_j)$. La mise à jour des contraintes actives se fait de la façon suivante :

$$\begin{cases} M_{i,j}(t) = M_{i,j} & \text{si } t = O_j + k \times L \times ppcm(T_i, T_j) \\ M_{i,j}(t) = M_{i,j}(t-1) & \text{sinon} \end{cases}$$

On peut observer alors qu'une contrainte de précédence $\tau_i \xrightarrow{M_{i,j},L} \tau_j$ est définie par rapport à $ppcm(T_i, T_j)$. La valeur du job en exécution de τ_i (resp. τ_j) par rapport à $M_{i,j}$ est définie comme $nb^i_{i\to j}(t) = \left\lfloor \frac{\max(0, T_i(t) - O_i(t))}{ppcm(T_i, T_j)} \right\rfloor$ (resp. $nb^j_{i\to j}(t) = \left\lfloor \frac{\max(0, T_j(t) - O_j(t))}{ppcm(T_i, T_j)} \right\rfloor$).

Exemple 23. *On exprime l'exécution temps réel de l'ensemble de tâches suivant avec une politique d'ordonnancement FP sur une architecture à deux processeurs.*

	O_i	T_i	D_i	C_i	$prio_i$
τ_0	0	5	5	1	1
τ_1	0	7	7	5	2
τ_2	0	10	10	7	3

$\mathcal{R}$

$\tau_0 \xrightarrow{\{(0,0),(3,2)\},1} \tau_1$

$\tau_2 \xrightarrow{\{(0,1)\},1} \tau_1$

L'exécution est décrite dans le tableau suivant :

t	0	1	2	5	7	8	10	15	16	35	70
τ_0	$(0,5,5,1)$	$(0,4,4,0)$	$(0,3,3,0)$	$(0,5,5,1)$	$(0,3,3,0)$	$(0,2,2,0)$	$(0,5,5,1)$	$(0,5,5,1)$	$(0,4,4,0)$	$(0,5,5,1)$	$(0,5,5,1)$
$nb^0_{0\to1}$	0	0	0	1	1	1	2	3	3	0	0
τ_1	$(0,7,7,5)$	$(0,6,6,5)$	$(0,5,5,4)$	$(0,2,2,1)$	$(0,7,7,5)$	$(0,6,6,5)$	$(0,4,4,3)$	$(0,6,6,5)$	$(0,5,5,5)$	$(0,7,7,5)$	$(0,7,7,5)$
$nb^1_{0\to1}$	0	0	0	0	1	1	1	2	2	0	0
$nb^1_{2\to1}$	0	0	0	0	1	1	1	2	2	5	0
$M_{0,1}(t)$ $M_{2,1}(t)$	$\{(0,0),(3,2)\}$ $\{(0,1)\}$	$\{(3,2)\}$ $\{(0,1)\}$	$\{(3,2)\}$ $\{(0,1)\}$	$\{(3,2)\}$ $\{(0,1)\}$	$\{(3,2)\}$ $\{(0,1)\}$	$\{(3,2)\}$	$\{(3,2)\}$	$\{(3,2)\}$		$\{(0,0),(3,2)\}$	$\{(0,0),(3,2)\}$ $\{(0,1)\}$
τ_2	$(0,10,10,7)$	$(0,9,9,6)$	$(0,8,8,5)$	$(0,5,5,2)$	$(0,3,3,1)$	$(0,2,2,0)$	$(0,10,10,7)$	$(0,5,5,3)$	$(0,4,4,2)$	$(0,5,5,3)$	$(0,10,10,7)$
$nb^2_{2\to1}$	0	0	0	0	0	0	1	1	1	3	0
$P(t)$	$\{0,2\}$	$\{1,2\}$	$\{1,2\}$	$\{0,1\}$	$\{2\}$	$\{1\}$	$\{0,1\}$	$\{0,2\}$	$\{1,2\}$	$\{0,2\}$	$\{0,2\}$

A l'instant 0*, seuls* τ^0_0 *et* τ^0_2 *peuvent s'exécuter car* τ^0_1 *est soumis à une contrainte de précédence* $\tau^0_0 \to \tau^0_1$*. A l'instant* 1*, le job* τ^0_0 *termine son exécution et libère la contrainte* $\tau^0_0 \to \tau^0_1$*. On enlève alors le mot* $(0,0)$ *de l'ensemble* $M_{0,1}(t)$*.* τ^0_1 *n'est plus soumis à aucune autre contrainte de précédence et peut donc démarrer son exécution. Les jobs* τ^0_2 *et* τ^0_1 *s'exécutent jusqu'à* 5 *où* τ^1_0 *se réveille. Comme cette tâche est la plus prioritaire, elle préempte le job* τ^0_2*. A* 6*,* τ^0_1 *et* τ^1_0 *terminent leur exécution et* τ^0_2 *reprend un processeur pour finir son exécution à 8. Le job* τ^1_1 *se réveille à* 7 *mais il est soumis à la contrainte de précédence* $\tau^0_2 \to \tau^1_0$*. Donc* τ^1_0 *ne peut s'exécuter tant que* τ^0_2 *n'est pas terminé.* τ^1_0 *devient donc libre de s'exécuter à 8. Le job* τ^2_1 *se réveille à 14 et est soumis à la contrainte de précédence* $\tau^3_0 \to \tau^2_1$*. Or le job* τ^3_0 *termine à 16 donc il retarde le démarrage de* τ^2_1*.*

Pour maintenir les valeurs des ensembles $M_{i,j}(t)$ *il faut utiliser des compteurs pour connaître le numéro du job en cours des tâches* τ_i *et* τ_j*. L'entier* $nb^0_{0\to1}$ *correspond au numéro du job de* τ_0 *pour la précédence* $\tau_0 \xrightarrow{\{(0,0),(3,2)\},1} \tau_1$*. Cet entier prend ses valeurs entre* $[0, ppcm(T_0, T_1) \times 1/T_0 - 1] = [0, 6]$*. De la même manière,* $nb^1_{0\to1}$ *correspond au numéro du job de* τ_1 *pour la même*

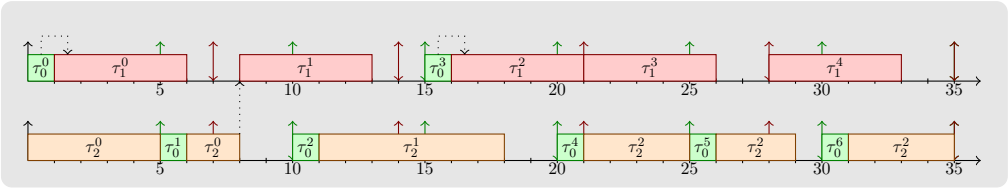


FIGURE 4.1 – Exécution sur deux processeurs et FP

précédence et prend ses valeurs entre $[0, ppcm(T_0, T_1) \times 1/T_1 - 1] = [0, 4]$. *A l'instant 35, les 2 entiers sont remis à 0 et l'ensemble* $M_{0,1}(35)$ *est réinitialisé à* $M_{0,1}$. *Les contraintes entre* τ_1 *et* τ_2 *sont traitées sur la période* $[0, 70]$. *Le diagramme de Gantt est donné figure 4.1.*

Exemple 24. *Considérons le même ensemble de tâches légèrement modifié (*$O_1 = 10$ *au lieu de* 0*), toujours exécuté avec une politique d'ordonnancement à priorité fixe sur une architecture à deux processeurs.*

	O_i	T_i	D_i	C_i	$prio_i$
τ_0	0	5	5	1	1
τ_1	10	7	7	5	2
τ_2	0	10	10	7	3

$$\mathcal{R} \quad \tau_0 \xrightarrow{\{(0,0),(3,2)\},1} \tau_1 \qquad \tau_2 \xrightarrow{\{(0,1)\},1} \tau_1$$

L'exécution est décrite dans le tableau suivant :

t	0	1	2	5	7	8	10	15	17	35	45
τ_0	$(0,5,5,1)$	$(0,4,4,0)$	$(0,3,3,0)$	$(0,5,5,1)$	$(0,3,3,0)$	$(0,2,2,0)$	$(0,5,5,1)$	$(0,5,5,1)$	$(0,3,3,0)$	$(0,5,5,1)$	$(0,5,5,1)$
$nb^0_{0\to1}$	0	0	0	1	1	1	2	3	3	0	2
τ_1	$(10,7,7,5)$	$(9,7,7,5)$	$(8,7,7,5)$	$(5,7,7,5)$	$(3,7,7,5)$	$(2,7,7,5)$	$(0,7,7,5)$	$(0,2,2,0)$	$(0,7,7,5)$	$(0,3,3,1)$	$(0,7,7,5)$
$nb^1_{0\to1}$	0	0	0	0	0	0	0	0	1	3	2
$nb^1_{2\to1}$	0	0	0	0	0	0	0	0	1	3	0
$M_{0,1}(t)$ $M_{2,1}(t)$	$\{(0,0),(3,2)\}$ $\{(0,1)\}$	$\{(3,2)\}$ $\{(0,1)\}$	$\{(3,2)\}$ $\{(0,1)\}$	$\{(3,2)\}$ $\{(0,1)\}$	$\{(3,2)\}$	$\{(3,2)\}$	$\{(3,2)\}$	$\{(3,2)\}$			$\{(3,2)\}$
τ_2	$(0,10,10,7)$	$(0,9,9,6)$	$(0,8,8,5)$	$(0,5,5,2)$	$(0,3,3,0)$	$(0,2,2,0)$	$(0,10,10,7)$	$(0,5,5,3)$	$(0,3,3,1)$	$(0,5,5,2)$	$(0,5,5,3)$
$nb^2_{2\to1}$	0	0	0	0	0	0	1	1	1	3	4
$P(t)$	$\{0,2\}$	$\{2\}$	$\{2\}$	$\{0,2\}$			$\{0,1\}$	$\{0,2\}$	$\{1,2\}$	$\{0,1\}$	$\{0,1\}$

Dans cet exemple, les contraintes de précédence n'influent pas le comportement car quand le job τ_1^0 *se réveille à* 10, *le job* τ_0^0 *a terminé depuis longtemps sa période et quand le job* τ_1^1 *se réveille,* τ_2^0 *a également terminé sa période. L'exécution est de faite équivalente à celle de l'ensemble de tâche* $\{\tau_0 = (0,5,5,1), \tau_1 = (10,7,7,5), \tau_2 = (0,10,10,7)\}$ *sans précédence.*

Propriété 1. *Soit le système composé deux tâches* $\mathcal{S} = \{\tau_i, \tau_j\}$ *et d'une contrainte de précédence* $\mathcal{R} = \tau_i \xrightarrow{\{(a,b)\},L} \tau_j$. *Alors*

$$O_i + aT_i + D_i \leq O_j + bT_j \implies (\mathcal{S}, \mathcal{R}) \equiv (\mathcal{S}, \emptyset)$$

Une contrainte de précédence est inutile si elle est vérifiée par construction.

Preuve 1. *Supposons que* $O_i + aT_i + D_i \leq O_j + bT_j$. *Cela implique que quand* τ_j^b *se réveille,* τ_i^a *est déjà terminée. Imposer la contrainte* $\tau_i^a \to \tau_j^b$ *est inutile car elle est respectée par construction.* $\tau_i \xrightarrow{\{(a,b)\},L} \tau_j$ *représente l'ensemble des contraintes* $\tau_i^{a+kLppcm(T_i,T_j)/T_i} \to \tau_j^{b+kLppcm(T_i,T_j)/T_j}$. *Or on a* $O_i + (a + kLppcm(T_i,T_j)/T_i)T_i + D_i = O_i + aT_i + D_i + kLppcm(T_i,T_j) \leq O_j + bT_j + kLppcm(T_i,T_j) = O_j + (b + kLppcm(T_i,T_j)/T_j)T_j$. *Donc toutes les contraintes sont vérifiées par définition des attributs temps réel de* τ_i *et* τ_j. *On en déduit que* $(\mathcal{S}, \mathcal{R})$ *et* $(\mathcal{S}, \emptyset)$ *ont le même comportement.* $\square$

On considère dans la suite des ensembles $(\mathcal{S}, \mathcal{R})$ tels que tous les mots de précédence sont "utiles", c'est-à-dire qu'ils ne sont pas vérifiés par construction. L'outil SchedMCore Converter applique la simplification si des contraintes sont inutiles.

Propriété 2. *Soit le système composé deux tâches $\mathcal{S} = \{\tau_i, \tau_j\}$ et d'une contrainte de précédence $\mathcal{R} = \tau_i \xrightarrow{\{(a,b)\},L} \tau_j$. Alors*

$$O_i + aT_i \geq O_j + bT_j + D_j \implies (\mathcal{S}, \mathcal{R}) \text{ n'est pas ordonnançable}$$

Si par construction on sait que le job τ_i^a se réveille toujours après la deadline de τ_j^b, il est impossible de respecter la contrainte de précédence $\tau_i^a \to \tau_j^b$.

Il est ensuite important de déterminer si un job est soumis ou non à une contrainte de précédence à un instant donné.

Définition 11. *Soit un système $(\mathcal{S}, \mathcal{R})$. Soit une tâche $\tau_i \in \mathcal{S}$ et soit l'ensemble des contraintes de précédence sur τ_i, à savoir $\tau_l \xrightarrow{M_{l,i},L_l} \tau_i$ avec $l = i_1, \ldots, i_p$. Le job τ_i^k doit s'exécuter après la fin des exécutions des jobs $\tau_{i_j}^{p_{i,j}^k}$ avec $(p_{i,j}^k, k) \in M'_{i_j,i}$. Les dépendances actives sur le job τ_i^k à l'instant t sont $prec(\tau_i^k, t) = \{(p_{i,j}^k, k \mod (ppcm(T_i, T_{i_j}) \times L/T_i)) \in M_{i_j,i}(t)\}$. Si $prec(\tau_i^k, t) = \emptyset$, alors le job n'est contraint par aucune précédence, on dit que le job est libre.*

Définition 12 (Tâche libre)**.** *Une tâche τ_i est dite* libre *à l'instant t, si le job en cours d'exécution à cet instant τ_i^k n'est soumis à aucune précédence :*

$$libre_i(t) = vrai \iff prec(\tau_i^k, t) = \emptyset \wedge O_i + kT_i \leq t < O_i + kT_i + D_i$$

Codage en C des contraintes de précédence Les contraintes de précédence sont stockées dans deux tableaux. Le premier `list_Dep_Var` stocke les valeurs des précédences à chaque instant t, tandis que le deuxième `list_Dep` stocke les valeurs initiales de chaque contrainte de précédence. Les deux tableaux ont une taille statique de $n \times PREC_TAILLE$ et sont identiques à l'instant 0. Ainsi, pour chaque tâche $\tau_i \in \mathcal{S}$, on stocke l'ensemble des contraintes de précédence sur τ_i, à savoir $\tau_{i_j} \xrightarrow{M_{i_j,i},L_{i_j}} \tau_i$ avec $j = 1, \ldots, p_i$. Le vecteur pour τ_i est

$$\langle p_i, i_1, L_{i_1} \times ppcm(T_{i_1}, T_i), \#M_{i_1,i}, M_{i_1,i}, \ldots, i_p, L_{i_p} \times ppcm(T_{i_p}, T_i), \#M_{i_p,i}, M_{i_p,i}, -1, \ldots, -1 \rangle$$

On met toutes les informations et on complète par des -1 pour obtenir une taille fixe. On a donc

$$PREC_TAILLE = \max_{i=1,\ldots,n} \left(1 + \sum_{k<=p_i} (2 + 2\#M_{i_k,i}) \right)$$

Exemple 25. *Pour le modèle de tâches de l'exemple 23 la déclaration des deux tableaux est dans l'algorithme 4.5.*

Seule τ_1 est soumise à des contraintes de précédence. Donc $PREC_TAILLE = 1+(2+4)+(2+2) = 13$. τ_0 et τ_2 n'ont pas de précédence, les vecteurs les concernant valent `0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1`. *Le vecteur de τ_1 est* `2,(0,35,2,(0,0),(3,2)),(2,70,1,(0,1))`. *La première valeur est le nombre de contraintes de précédence. Ensuite, le vecteur se divise en deux groupes : les données pour la première contrainte* `(0,35,2,(0,0),(3,2))` *où la première valeur*

Algorithme 4.5: Déclaration des précédences

```
1 const int PREC_TAILLE = 13
2 const int list_Dep[TASK_NUMBER][PREC_TAILLE]={ {0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
3                                                {2,0,35,2,0,0,3,2,2,70,1,0,1},
4                                                {0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}}
5 int list_Dep_Var[TASK_NUMBER][PREC_TAILLE]={ {0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1},
6                                              {2,0,35,2,0,0,3,2,2,70,1,0,1},
7                                              {0,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1}}
```

"0" indique la tâche prédécesseur, la deuxième "35" le ppcm entre les périodes de τ_1 et τ_0, la troisième "2" le nombre de mots de la contrainte et ensuite les deux mots "`(0,0),(3,2)`"; le deuxième groupe suit le même principe pour la deuxième contrainte temporelle, ainsi la tâche prédécesseur est "2", le ppcm de périodes "70" et il a qu'un seule mot "`(0,1)`".

Il faut ensuite maintenir les informations sur les numéros de jobs en cours. Pour cela, on utilise autant de variables que de contraintes de précédence. On stocke dans `job_i_to_j_i` stocke le numéro du job τ_i en cours et `nb_job_i_to_j_i` stocke le nombre de jobs par répétition. C'est équivalent des entiers $nb^i_{i\to j}$ de l'exemple 23. Chaque $nb^i_{i\to j}$ prend ses valeurs dans $[0, ppcm(T_i, T_j) \times L/T_i - 1]$.

Exemple 26. *Pour le modèle de tâches de l'exemple 23 la déclaration des deux tableaux est donnée dans l'algorithme 4.6.*

Algorithme 4.6: Déclaration des numéros de jobs

```
1 const int nb_job_0_to_1_0=7
2 const int nb_job_0_to_1_1=5
3 const int nb_job_2_to_1_2=7
4 const int nb_job_2_to_1_1=10
5
6 int job_0_to_1_0=0
7 int job_0_to_1_1=0
8 int job_2_to_1_2=0
9 int job_2_to_1_1=0
```

`job_0_to_1_0` comme $nb^1_{0\to 1}$ compte le numéro du job sur l'intervalle $[0, 35]$. Initialement, tous les jobs commencent au numéro 0.

Codage en C de l'évolution des contraintes de précédence L'opération

$$M_{i,j}(t) = M_{i,j}(t-1) \setminus \{(m, m')\} \text{ si } t = e_{\tau_i}(m + k\frac{pL}{T_i})$$

se fait par la fonction `updateDep(ti,tj)` 4.7. A la fin de l'exécution d'un job de τ_i, il faut supprimer toutes les dépendances liées à ce job sur τ_j. Cela se fait en mettant les valeurs $(-1, -1)$ sur le mot correspondant.

L'opération

$$M_{i,j}(t) = M_{i,j} \text{ si } t = O_j + k \times L \times ppcm(T_j, T_j)$$

se fait par la fonction `initiateDep(ti,tj)` 4.8. Une fois qu'une tâche τ_i a terminé une période de longueur $ppcm(T_i, T_j) \times L/T_j$, il faut réinitialiser correctement les précédences. Comme pour la fonction `updateDep(ti,tj)`, on parcourt les précédences jusqu'aux mots reliant τ_i et τ_j. Cette fois, on remet les valeurs initiales.

Fonction 4.7 : updateDep(ti,tj)

```
1  pos⟵ 0;
2  pour i ← 0 to list_Dep_Var[tj][0] faire
3      pos⟵ pos+2;
4      nbWord⟵ list_Dep_Var[tj][pos+1];
5      si list_Dep_Var[tj][pos-1]==ti alors
6          pour j ← 0 to nbWord faire
7              pos⟵ pos+2;
8              si list_Dep_Var[tj][pos]==job_i_to_j_i alors
9                  list_Dep_Var[tj][pos]⟵ −1;
10                 list_Dep_Var[tj][pos+1]⟵ −1;
11     sinon
12         pos=pos+2*nbWord ;
13     pos++ ;
```

Fonction 4.8 : initiateDep(ti,tj)

```
1  pos⟵ 0;
2  pour j ← 0 to list_Dep_Var[tj][0] faire
3      pos⟵ pos+2;
4      nbWord⟵ list_Dep_Var[tj][pos+1];
5      si list_Dep_Var[tj][pos-1]==ti alors
6          pour j ← 0 to nbWord faire
7              pos⟵ pos+2;
8              list_Dep_Var[tj][pos]⟵list_Dep[tj][pos];
9              list_Dep_Var[tj][pos+1]⟵list_Dep[tj][pos+1];
10     sinon
11         pos=pos+2*nbWord ;
12     pos++ ;
```

Pour coder la notion de libre, on utilise la fonction `isTaskFree(task)` 4.9 qui teste si une tâche, à un instant donnée, a des contraintes de précédence à respecter. Lorsque `isTaskFree(task)` =*vrai*, le job en cours d'exécution est libre sinon il est soumis à au moins une contrainte de précédence.

Fonction 4.9 : isTaskFree(task)

```
1  pos ⟵ 0;
2  pour i ← 0 to list_Dep_Var[task][0] faire
3      pos = pos +3;
4      ti=pos;
5      nbWord = list_Dep_Var[task][pos];
6      pour j ← 0 to nbWord faire
7          pos = pos +2;
8          si list_Dep_Var[task][pos] = job_ti_to_task_task alors
9              retourner false
10 retourner true
```

On stocke les valeurs de liberté dans un tableau `free_Task`.

Exemple 27. *Pour le modèle de tâches de l'exemple 23 la déclaration est donnée dans l'algorithme 4.10.*

Pour finir avec la gestion de contraintes de précédences, la méthode `step()` 4.11 est légèrement modifiée. En effet, ligne 9, quand une tâche termine son exécution, on relâche toutes les

Algorithme 4.10: Déclaration des jobs libres

```
1  int free_Task[TASK_NUMBER]={true,false,true}
2
3  void mise_a_jour_free() {
4    int i
5      for (i=0 ; i< TASK_NUMBER ; i ++)
6        free_task[i] = isTaskFree(i)
7  }
```

contraintes la concernant en appelant `updateDep`. Ligne 17, on met à jour les numéros de jobs concernant la tâche i, si un des numéros passe à 0, on réinitialise les précédences en appelant `initiateDep`. Il existe également une version optimisée avec les précédences que nous détaillons pas ici.

Fonction 4.11 : step

```
1  pour i ← 0 to TASK_NUMBER faire
2      si list_o[i]=0 alors
3          si list_t_var[i] > 0 alors
4              list_t_var[i]–;
5          si list_d_var[i] > 0 alors
6              list_d_var[i]–;
7          si prio(i) alors
8              list_c_var[i]–;
9              si list_c_var[i]=0 alors
10                 pour j ← 0 to TASK_NUMBER faire
11                     updateDep(i,j) ;
12         si list_t_var[i]=0 alors
13             si list_c_var[i]=0 alors
14                 list_t_var[i]=list_t[i];
15                 list_w_var[i]=list_w[i];
16                 list_d_var[i]=list_d[i];
17                 pour j ← 0 to TASK_NUMBER faire
18                     job_i_to_j_i =(job_i_to_j_i+1)  mod  nb_job_i_to_j_i;
19                     si job_i_to_j_i=0 alors
20                         initiateDep(j,i);
21     sinon
22         list_o[i] –;
```

4.2 Analyse d'ordonnançabilité par parcours exhaustif

Nous avons choisi d'encoder l'analyse d'ordonnançabilité par un parcours exhaustif. Dans la section précédente, nous avons décrit le codage des configurations et l'avancement temporel par la fonction `step`. Il nous faut maintenant préciser le codage de la fonction `prio` et la manière d'avancer dans la simulation.

4.2.1 Encodage des politiques d'ordonnancement

L'objectif de cette partie est de montrer l'évolution de $P(t)$ en fonction du temps et donc l'encodage de la fonction `prio` utilisée dans la fonction `step`, ligne 7 des fonctions 4.2 et 4.11, ligne 6 pour la fonction 4.4. En effet, $P(t)$ et `prio` encodent les choix de la politique d'ordonnancement.

Elles trient les tâches actives en fonction des priorités à l'instant t. Une tâche τ_i est active si elle est réveillée ($O_i(t) = 0$), elle n'est pas terminée ($C_i(t) > 0$) et elle est libre de toute précédence ($libre_i(t)$= $vrai$).

$P(t)$ est encodé par une liste `list_prio` qui stocke à tout moment les tâches par ordre de priorités : la tâche la plus prioritaire est dans la première position, la deuxième dans la deuxième et ainsi de suite.

Exemple 28. *Considérons l'ensemble de tâche sans précédence* $\{\tau_0 = (0, 5, 5, 2), \tau_1 = (1, 5, 5, 5), \tau_2 = (1, 5, 5, 2)\}$ *de l'exemple 20 avec une politique d'ordonnancement à priorité fixe RM sur une architecture à deux processeurs. On a alors la déclaration suivante :*

Algorithme 4.12: Déclaration des priorités initiales

```
1 int list_prio[TASK_NUMBER]= {0 , 1 , 2}
```

On remarque alors que $P(t)$ est l'ensemble des premiers éléments de `list_prio`. Il faut ensuite mettre à jour $P(t)$ et donc `list_prio` au cours du temps. Ceci sera fait par la fonction `UpdatePrio`.

On considère à nouveau la fonction $e_{\tau_i}(k) : \mathbb{N} \to \mathbb{N}$ où $e_{\tau_i}(k)$ est l'instant de finalisation du job τ_i^k. On rappelle que cela signifie que $C_i(t) = 0$, $C_i(t-1) = 1$ et $t = e_{\tau_i}(k)$. A l'instant t, $P(t)$ contient toujours moins d'éléments que le nombre de tâches actives et le nombre de processeurs. La fonction $less_prio(P(t))$ renvoie l'indice de la tâche la moins prioritaire de l'ensemble $P(t)$. Alors, à tout instant t, si $l \in P(t)$ et $l \neq less_prio(P(t))$, on a $\tau_l < \tau_{less_prio(P(t))}$ au sens des priorités. L'évolution des tâches actives prioritaires se fait en deux étapes :

- quand une tâche de $P(t-1)$ termine son exécution à t, c'est-à-dire $t = e_{\tau_l}(k)$, alors la tâche est retirée de l'ensemble

$$\begin{cases} P(t) = P(t-1) \setminus \{l\} & \text{si } t = e_{\tau_l}(k) \\ P(t) = P(t-1) & \text{sinon} \end{cases}$$

- quand une tâche prioritaire se réveille, elle est ajoutée dans $P(t)$. Si le cardinal de P est inférieur strictement au nombre de processeur, elle est simplement ajoutée, sinon, il faut retirer la tâche la moins prioritaire de l'ensemble

$$\begin{cases} P(t) = P(t) \cup \{l\} & \text{si } t = O_l + kT_l \text{ et } \#P(t) < nb_proc \\ P(t) = (P(t) \setminus \{less_prio(P(t))\}) \cup \{l\} & \text{si } t = O_l + kT_l \text{ et } \#P(t) = nb_proc \text{ et } \tau_l < \tau_{less_prio(P(t))} \\ P(t) = P(t) & \text{sinon} \end{cases}$$

Ce qui différencie les politiques d'ordonnancement est la manière de calculer si $\tau_l < \tau_{less_prio(P(t))}$ à l'instant t.

Codage de FP

La politique FP est la plus simple car la liste des priorités est constante.

Codage de gEDF et gLLF

Dans le cas de gEDF et gLLF, il faut modifier la liste au fur et à mesure de l'exécution. Commençons par la politique gEDF. Le codage en C se fait alors avec la fonction *updatePrio* 4.13.

Fonction 4.13 : updatePrio pour gEDF

```
1  k⟵ 0 ;
2  pour i ← 0 to TASK_NUMBER faire
3      pour j ← i + 1 to TASK_NUMBER faire
4          si list_o[list_prio[j]]=0 ∧ list_c_var[list_prio[j]]>0 ∧ isTaskFree (list_prio[j]) alors
5              si list_o[list_prio[i]]>0 ∨ list_c_var[list_prio[i]]=0 ∨¬ isTaskFree (list_prio[i]) alors
6                  aux⟵ list_prio[i];
7                  list_prio[i]⟵ list_prio[j];
8                  list_prio[j]⟵ aux;
9              sinon si list_d_var[list_prio[j]]<list_d_var[list_prio[i]] alors
10                 aux⟵ list_prio[i];
11                 list_prio[i]⟵ list_prio[j];
12                 list_prio[j]⟵ aux;
```

L'idée est la suivante : on parcourt la liste dans l'ordre, si la tâche τ_i n'est pas active et si on trouve une tâche stockée plus loin τ_j active, alors on les permute (ligne 5). Si la tâche τ_i est active et si on trouve une tâche stockée plus loin τ_j active et plus prioritaire, on les permute (ligne 9). On maintient ainsi la liste à jour au fur et à mesure de l'exécution.

La condition de la ligne 9 est différente selon la politique, on a :

- pour gEDF : *list_d_var[list_prio[j]]<list_d_var[list_prio[i]]*
- pour gLLF : *(list_d_var[list_prio[j]] - list_c_var[list_prio[j]]) < (list_d_var[list_prio[i]] - list_c_var[list_prio[i]])*

Codage de LLREF

Le cas de LLREF est un peu plus complexe. Nous avons rencontré deux difficultés pour coder cette politique :

1. les paramètres sont exprimés pour des tâches synchrones à échéances implicites,
2. les paramètres ont des valeurs réelles. Or nous avons choisi un pas de temps discret et Uppaal ne permet que de manipuler des entiers.

Pour le premier problème, nous avons étendu les définitions pour des ensembles de tâches asynchrones aux échéances contraintes. Pour le deuxième point, nous avons fait une modification pour réaliser des pas d'exécution entiers.

Extension à un ensemble de tâches asynchrones aux échéances contraintes Le politique LLREF a été définie originalement pour des ensembles de tâches synchrones aux échéances implicites. Pour prendre en compte les dates de réveil, nous modifions le calcul de T' en tenant compte de ces dates. Pour prendre en compte les échéances, nous transformons la formule sur le taux d'exécution équitable de la formule 2.1

$$l_\tau(t) = C_\tau(t) \times \frac{T' - t}{T_\tau - t}$$

en

$$l_\tau(t) = C_\tau(t) \times \frac{T' - t}{D_\tau - t} \tag{4.1}$$

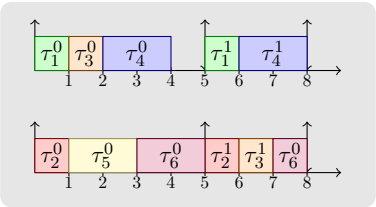
Exemple 29. *Considérons l'ensemble de tâches du tableau suivant, ordonnançable avec une politique à priorité fixe avec $\tau_1 < \tau_3 < \tau_2 < \tau_4 < \tau_5 < \tau_6$ sur deux processeurs.*

τ_i	O_i	T_i	D_i	C_i
τ_1	0	5	1	1
τ_2	0	5	2	1
τ_3	0	5	1	1
τ_4	0	5	3	2
τ_5	0	8	5	2
τ_6	0	8	7	2

Si on applique la formule standard 2.1 de LLREF ne prenant en compte que les périodes, on a les valeurs du tableau 4.2(a) et l'exécution du chronogramme 4.2(b).

t	0	1	2	3	4	5	6	7	...
T'	1	2	3	5	5	6	7	8	...
$l_1(t)$	1/5	1/4	-	-	-	1/5	-	-	...
$L_1(t)$	4/5	3/4	-	-	-	4/5	-	-	...
$l_2(t)$	1/5	1/4	1/3	-	-	1/5	1/4	-	...
$L_2(t)$	4/5	3/4	2/3	-	-	4/5	3/4	-	...
$l_3(t)$	1/5	1/4	1/3	-	-	1/5	1/4	-	...
$L_3(t)$	4/5	3/4	2/3	-	-	4/5	3/4	-	...
$l_4(t)$	2/5	1/4	1/3	1	-	2/5	1/4	1/3	...
$L_4(t)$	3/5	3/4	2/3	1	-	3/5	3/4	2/3	...
$l_5(t)$	1/4	1/7	1/6	1/5	-	-	-	-	...
$L_5(t)$	3/4	6/7	5/6	9/5	-	-	-	-	...
$l_6(t)$	1/4	1/3	1/5	1/2	-	-	-	-	...
$L_6(t)$	3/4	2/3	4/5	3/2	-	-	-	-	...

(a)



(b)

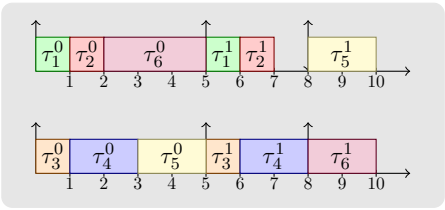
FIGURE 4.2 – Exécution en LLREF

Il y a plusieurs erreurs temporelles dans cette exécution. Les jobs τ_1^0 et τ_3^0 doivent s'exécuter immédiatement après leur activation car leur échéance est identique. Malheureusement, si on calcule la priorité par rapport à la période uniquement, l'urgence n'est pas détectée et on exécute τ_4^0 et τ_5^0 qui ont une laxité supérieure. Cette erreur se reproduit plusieurs fois dans cet exemple, τ_4^0 termine à 4, etc.

En revanche, si on utilise la formule modifiée 4.1 l'exécution se développe correctement comme montré dans le tableau 4.3(a) et le chronogramme 4.3(b).

t	0	1	2	3	4	5	6	7	8	...
T'	1	2	3	5	5	6	7	8	10	...
$l_1(t)$	1	-	-	-	-	1	-	-	-	...
$L_1(t)$	0	-	-	-	-	0	-	-	-	...
$l_2(t)$	1/2	1	-	-	-	1/2	1	-	-	...
$L_2(t)$	1/2	0	-	-	-	1/2	0	-	-	...
$l_3(t)$	1	-	-	-	-	1	-	-	-	...
$L_3(t)$	0	-	-	-	-	0	-	-	-	...
$l_4(t)$	2/3	1	1	-	-	2/3	1	-	-	...
$L_4(t)$	1/3	0	0	-	-	1/3	0	-	-	...
$l_5(t)$	2/5	1/2	2/3	1	-	-	-	-	4/5	...
$L_5(t)$	3/5	1/2	1/3	1	-	-	-	-	6/5	...
$l_6(t)$	2/7	1/3	2/5	1	1/3	-	-	-	4/7	...
$L_6(t)$	5/7	2/3	3/5	1	2/3	-	-	-	10/7	...

(a)



(b)

FIGURE 4.3 – Exécution en LLREF modifié

Passage des paramètres à valeurs réelles à des paramètres à valeurs entières La politique d'ordonnancement LLREF utilise des valeurs dans $\mathbb{R}$ pour les paramètres $l_i(t)$ et $L_i(t)$.

Malheureusement, UPPAAL n'est pas capable d'utiliser des variables à valeurs réelles. Pour cela, nous avons décomposé les paramètres l et L en deux parties : un entier pour stocker le quotient et un autre pour le reste de la division euclidienne. Si δ_{cond} est la fonction égale à 1 si *cond* est *vrai* et 0 sinon. Les paramètres deviennent :

$$\begin{array}{l} l_i^q(t) = (\delta_{O_i(t)=0} \times C_i(t) \times T')/D_i(t) \\ l_i^r(t) = (\delta_{O_i(t)=0} \times C_i(t) \times T') \ mod \ D_i(t) \\ L_i^q(t) = T' - l_i^q(t) \end{array} \tag{4.2}$$

Le paramètre $L_\tau^r(t)$ n'est pas nécessaire pour déterminer la priorité car une tâche est urgente lorsque $L_\tau(t) = 0$ et cela implique $L_\tau^q(t) = 0$. Par conséquence, nous ne calculons pas cette variable.

Exemple 30. *Considérons l'ensemble de tâche de l'exemple 29 et appliquons la stratégie expliquée :*

t	0	1	2	3	4	5	6	7	8	...
T'	1	2	3	5	5	6	7	8	10	...
$l_1^q(t)$	1	-	-	-	-	1	-	-	-	...
$l_1^r(t)$	0	-	-	-	-	0	-	-	-	...
$L_1^q(t)$	0	-	-	-	-	0	-	-	-	...
$l_2^q(t)$	0	1	-	-	-	0	1	-	-	...
$l_2^r(t)$	1	0	-	-	-	1	0	-	-	...
$L_2^q(t)$	1	0	-	-	-	1	0	-	-	...
$l_3^q(t)$	1	-	-	-	-	1	-	-	-	...
$l_3^r(t)$	1	-	-	-	-	1	-	-	-	...
$L_3^q(t)$	0	-	-	-	-	0	-	-	-	...
$l_4^q(t)$	0	1	1	-	-	0	1	-	-	...
$l_4^r(t)$	2	1	1	-	-	2	1	-	-	...
$L_4^q(t)$	1	0	0	-	-	1	0	-	-	...
$l_5^q(t)$	0	0	0	1	-	-	-	-	0	...
$l_5^r(t)$	5	1	2	1	-	-	-	-	4	...
$L_5^q(t)$	1	1	1	1	-	-	-	-	2	...
$l_6^q(t)$	0	0	0	1	0	-	-	-	0	...
$l_6^r(t)$	2	1	2	1	1	-	-	-	4	...
$L_6^q(t)$	1	1	1	1	1	-	-	-	2	...

Codage de la fonction `updatePrio` La fonction `updatePrio()` 4.14 calcule la priorité des tâches et la prochaine date de réveil à un instant t. D'abord, la première boucle (ligne 3) calcule la prochaine date T' stockée dans la variable `nexttime`. Pour cela, on prend le minimum entre les prochaines dates de réveil, échéances et périodes. La boucle suivante (ligne 10) est responsable de trier la liste `list_prio` en fonction des critères de LLREF. La première condition (ligne 13) indique qu'on déplace les tâches non actives. Ensuite, si les deux tâches τ_i et τ_j sont actives, on les trie ligne 17. Pour cela, on choisit les tâches urgentes `LQ=0`, puis celles avec les `lq` les plus grands. En cas d'égalité sur les `lq`, les tâches les plus prioritaires sont les tâches avec les `lr` les plus petits. Il faut également calculer t' qui est sauvé dans la variable `tnext`.

4.2.2 Encodage de l'exploration

Cette partie décrit la manière dont est réalisé le parcours exhaustif. Il faut simuler l'exécution du système comme illustré dans les différents exemples et notamment l'exemple 19 page 51. Comme nous ne connaissons pas l'intervalle de faisabilité, nous arrêtons l'exploration dès qu'un état a déjà été visité. Nous avons proposé deux analyses :

1. une simulation déterministe où le choix est unique pour une configuration donnée. Nous avons implanté cette méthode en C et en UPPAAL,

Fonction 4.14 : updatePrio pour LLREF

```
1  k⟵ 0;
2  nexttime⟵list_t_var[0];
3  pour i ← 0 to TASK_NUMBER faire
4      si list_o[i]<nexttime ∧ list_o[i]>0 alors
5          nexttime⟵ list_o[i];
6      sinon si list_d_var[i]<nexttime ∧ list_d_var[i]>0 alors
7          nexttime⟵ list_d_var[i];
8      sinon si list_t_var[i]<nexttime alors
9          nexttime⟵ list_t_var[i];
10 pour i ← 0 to TASK_NUMBER faire
11     pour j ← i + 1 to TASK_NUMBER faire
12         si list_o[list_prio[j]]=0 ∧ list_c_var[list_prio[j]]>0 ∧ isTaskFree (list_prio[j]) alors
13             si list_o[list_prio[i]]>0 ∨ list_c_var[list_prio[i]]=0 ∨¬ isTaskFree (list_prio[i]) alors
14                 aux⟵list_prio[i];
15                 list_prio[i]⟵list_prio[j];
16                 list_prio[j]⟵aux;
17             sinon si (LQ[list_prio[j]]=0 ∧ LQ[list_prio[i]]>0) ∨ (LQ[list_prio[i]]>0 ∧ LQ[list_prio[j]]>0 ∧
               ((lq(list_prio[j]) > lq(list_prio[i]) ∨ (lq(list_prio[i]) = lq(list_prio[j] ∧ lr(list_prio[i])×
               list_d_var[list_prio[j]] < lr(list_prio[j]) × list_d_var[list_prio[i]])))) alors
18                 aux⟵list_prio[i];
19                 list_prio[i]⟵list_prio[j];
20                 list_prio[j]⟵aux;
```

2. une simulation non déterministe. Si la politique associe la même priorité à deux tâches distinctes à un instant t, nous considérons l'ensemble des exécutions engendrées pour les deux ordres possibles. Un exemple de simulation non déterministe est donné dans l'exemple 31. Cette méthode n'est implantée qu'en Uppaal.

Exemple 31. *Soit l'ensemble de tâches* $\mathcal{S} = \{\tau_1 = (0,5,5,2), \tau_2 = (0,5,5,2), \tau_3 = (0,5,5,2)\}$ *si on l'exécute sur une architecture à deux processeurs avec gEDF, toutes les tâches ont la même priorité car l'échéance est la même. Alors, plusieurs exécutions peuvent être réalisées, comme montré dans la figure 4.4.*

La recherche indéterministe offre la possibilité d'explorer toutes les exécutions possibles d'un système. Cependant, dans le cas d'une exécution réelle sur une architecture et avec un système d'exploitation donnés, ce non déterminisme est inexistant. En effet, toutes les plateformes d'exécution ont un comportement déterministe : même si deux tâches ont théoriquement la même priorité, l'ordonnanceur privilégie toujours une des deux. En conséquence, une vérification déterministe conforme à l'exécutif réel est suffisante.

Finitude de l'espace d'état

Les explorations aboutissent car le nombre d'états est fini. En effet, nous regardons les exécutions avec des changements d'états à date entière et tous les paramètres sont exprimés en temps relatif. Les politiques d'ordonnancement considérées sont conservatives et sans mémoire, c'est-à-dire qu'elles calculent les priorités en fonction de la configuration courante.

Nous montrons dans le premier lemme que les paramètres de période et d'échéance sont périodiques de période l'hyper-période H.

Lemme 1. *Soit l'hyper-période* $H = \text{ppcm}_{i \leq n}(T_i)$. $\forall i \in [1,n]$ *et* $\forall t \geq max_{i \leq n}(O_i)$, *nous avons :*

$$T_i(t+H) = T_i(t) \ \wedge \ D_i(t+H) = D_i(t)$$

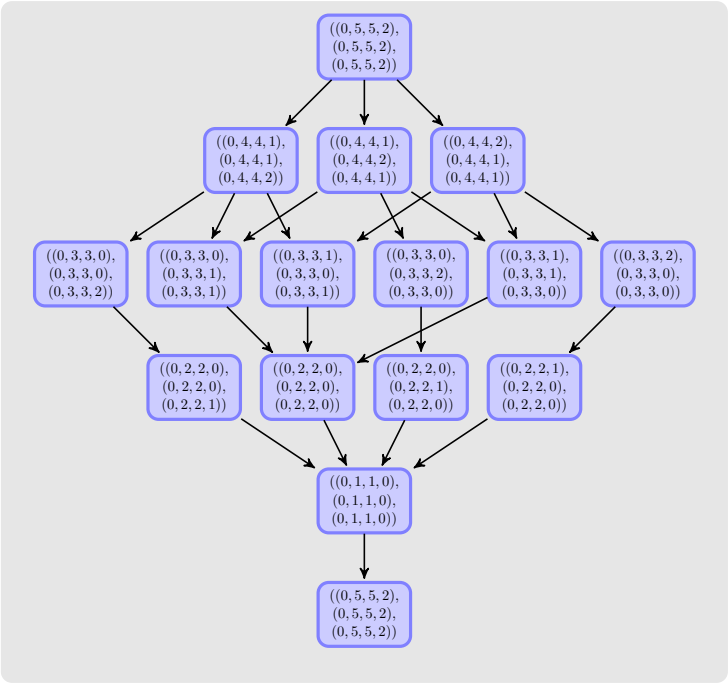


FIGURE 4.4 – Exploration non déterministe

Preuve 2. *La preuve découle de la définition de $T_i(t)$. En effet, si $t \geq max_{i \leq n}(O_i)$ et $i \in [1, n]$, on a $T_i(t) = T_i + ((t - O_i) \ mod \ T_i)$. Donc, $T_i(t + H) = T_i + ((t + H - O_i) \ mod \ T_i) = T_i(t)$ puisque $H \ mod \ T_i = 0$. On a de plus, $D_i(t) = max(0, T_i(t) - (T_i - D_i))$, donc $D_i(t)$ est également périodique.* □

Propriété 3. *Il y a un nombre fini d'états dans l'exécution d'un système. Il est suffisant d'étudier au plus la fenêtre $[0, max_{i \leq n}(O_i) + p \times H]$ où $p = \prod_i (C_i + 1)$.*

Preuve 3. *Le problème se ramène à un problème d'énumération. A chaque instant $t \geq max_{i \leq n}(O_i)$ tel que $t - max_{i \leq n}(O_i) \equiv 0 \ mod \ (H)$, on sait d'après le lemme précédent que $O_i(t) = 0$, $T_i(t)$ et $D_i(t)$ sont constants. Le seul paramètre qui peut changer est $C_i(t) \in [0, C_i]$. Comme $C_i(t)$ ne peut prendre que $C_i + 1$ valeurs, on en déduit qu'il y a au plus $p = \prod_i (C_i + 1)$ états différents aux instants t.* □

En faisant l'analyse d'ordonnançabilité pour une des politiques (FP, gEDF, gLLF ou LLREF), les choix sont identiques à partir d'une même configuration. L'idée est alors (1) de vérifier que les exécutions jusqu'en $max_{i \leq n}(O_i) + H$ sont correctes ; (2) de chercher une configuration qui se répète à partir de l'instant $max_{i \leq n}(O_i)$ et ce, aux instants $t - max_{i \leq n}(O_i) \equiv 0 \ mod \ (H)$. Pour le point (2), on regarde à l'instant $max_{i \leq n}(O_i) + H$ si $\forall i$, $C_i(max_{i \leq n}(O_i) + H) = C_i(max_{i \leq n}(O_i))$. Si tel est le cas, l'exploration est terminée, si non on explore jusqu'à $max_{i \leq n}(O_i) + 2H$ et on regarde si $\forall i$, $C_i(max_{i \leq n}(O_i) + 2H) = C_i(max_{i \leq n}(O_i))$ ou si $\forall i$, $C_i(max_{i \leq n}(O_i) + 2H) = C_i(max_{i \leq n}(O_i) + H)$. Et ainsi de suite jusqu'à trouver une configuration stable. Au maximum, on aura besoin de $\prod_i (C_i + 1)$ pas pour compléter l'exploration.

Nous sommes donc assurés qu'une exploration exhaustive trouvera toujours une solution. En pratique, l'implantation est déterminante car il faut éviter le problème de l'explosion combinatoire. Dans le cas déterministe, le version C est plus efficace, car nous sauvons les états toutes les H unités de temps et le code aboutit toujours.

Exploration déterministe

L'exploration déterministe consiste à appliquer la simulation en choisissant de manière fixe un ordre quand deux tâches ont la même priorité.

Exploration en UPPAAL se modélise par un automate à deux états. Dans l'état `ok`, le temps avance de manière discrète grâce à la transition `updateTime()` qui correspond à un pas de temps. L'effet est d'appliquer la fonction `updateTime` sur les variables du programme. L'automate reste donc dans l'état `ok` tant qu'aucune contrainte de temps ou de précédence n'est violée. L'état `ko` correspond à un problème dans l'exécution. Si cet état est atteignable, alors l'ensemble de tâches n'est pas ordonnançable.

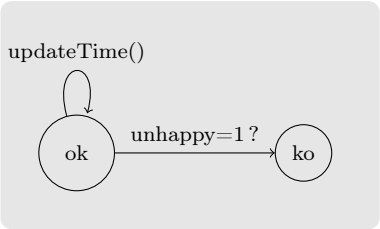


FIGURE 4.5 – Exécution in UPPAAL

Fonction 4.15 : Fonction updateTime

```
1 step();
2 pour i ← 0 to TASK_NUMBER faire
3     si list_o[i]=0 et list_c_var[i] > list_d_var[i] alors
4         unhappy = 1;
5     si changerPrio = 1 alors
6         updatePrio();
```

La fonction `updateTime()` 4.15 applique dans un premier temps la fonction `step`, puis évalue ensuite si une condition temporelle est invalidée, ce qui passera variable `unhappy` à 1. Le booléen `changerPrio` dépend de la politique d'ordonnancement utilisée. Ainsi, pour FP ou gEDF la priorité ne doit être mise à jour que si un job démarre ou se termine. Dans gLLF on doit recalculer la priorité à chaque pas de temps. La mise à jour de LLREF dépend des variables intermédiaires `nexttime` et `tnext`.

La vérification de l'ordonnançabilité se fait ensuite en appelant le model checker et en vérifiant la propriété d'accessibilité :

$$A[]not\ ko$$

Exploration en C La programmation en C se fait par la fonction `main` 4.16. Quand l'ensemble de tâches est synchrone, il suffit de simuler jusqu'à l'hyper-période.

Si l'ensemble de tâche est asynchrone, on sauve le tableau `list_c_var` toutes les hyper-périodes longueurs de temps à partir de $\max(O_i)$. L'algorithme est montré en 4.17.

Algorithme 4.16: Fonction principale main() pour ensemble de tâches synchrones

```
1  unhappy = 0;
2  tant que instant < H et unhappy=0 faire
3      updateTime() ; instant ++;
4  si unhappy=0 alors
5      imprimer("ensemble de tâches ordonnançable")
6  sinon
7      imprimer("ensemble de tâches non ordonnançable")
```

Algorithme 4.17: Fonction principale main() pour ensemble de tâches asynchrones

```
1  unhappy = 0;
2  cyclic = false;
3  while !cyclic et unhappy=0 do
4      updateTime();
5      si (instant-MAXO)%H==0 et instant>0 alors
6          cnt=0;
7          while cnt<TASK_NUMBER do
8              si list_c_var[cnt]=w_save[cnt] alors
9                  cnt++;
10         si cnt=TASK_NUMBER alors
11             cyclic=true;
12         sinon
13             pour j=0 to TASK_NUMBER faire
14                 w_save[j]=list_c_var[j];
15     instant ++;
16 si unhappy=0 alors
17     imprimer("ensemble de tâches ordonnançable")
18 sinon
19     imprimer("ensemble de tâches non ordonnançable")
```

Exploration non déterministe

L'automate associé à l'exploration non déterministe est montré dans l'image 4.6. L'automate comporte toujours les 2 états `ok` et `ko`, ainsi que la transition temporelle `updateTime`. La différence réside dans la transition étiquetée par *swap_priority()* : cette boucle génère l'indéterminisme, c'est-à-dire force l'exploration de tous les ordres possibles en cas d'égalité de priorité. Lorsque la boucle est tirée, on réalise l'action *swap_priority()*. Cette fonction intervertit l'ordre de deux tâches avec la même priorité à l'instant t dans `list_prio`. Le code est illustré dans la fonction 4.18. Pour le cas de FP nous considérons des priorités uniques, donc cette situation ne se produit jamais.

Fonction 4.18 : swapPriority(i,j)

```
1  p ⟵ 0 ;
2  k ⟵ 1 ;
3  tant que (listPrio[k] ≠ i ∧ listPrio[k] ≠ j) faire
4      k ⟵ k + 1;
5      tant que (listPrio[k+p]≠ i ∧ listPrio[k+p] ≠ j) faire
6          p ⟵ p + 1 ;
7          temp ⟵listPrio[k];
8          listPrio[k] ⟵ listPrio[k+p];
9          listPrio[k+p] ⟵ temp;
```

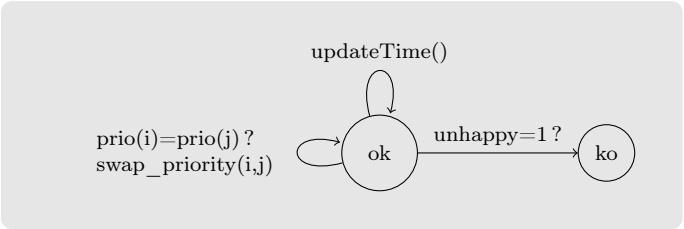


FIGURE 4.6 – Execution in UPPAAL

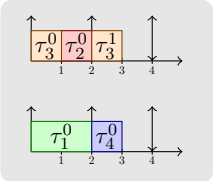
Dans le cas d'une exploration non déterministe, il se peut que certaines exécutions soient ordonnançables et d'autres non. On considère qu'un système est ordonnançable de manière non déterministe, si toutes les séquences possibles respectent les contraintes temporelles et de précédence. Comme dans le cas déterministe la vérification de l'ordonnançabilité se fait en vérifiant la propriété :

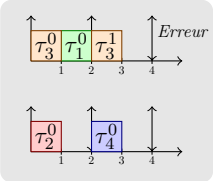
$$A[]not\ ko$$

Propriété 4 (Erratum)**.** *La transition* swap_priority() *est nécessaire pour gLLF contrairement à notre intuition [CBNP11].*

Preuve 4. *Pour l'ensemble de tâches suivant exécuté sur deux processeurs :*

τ_i	O_i	T_i	D_i	C_i
τ_1	0	4	3	2
τ_2	0	4	2	1
τ_3	0	2	1	1
τ_4	2	4	1	1





Les deux exécutions possibles sont illustrées dans les chronogrammes. A l'initialisation de l'ordonnancement, trois tâches peuvent être exécutées, τ_1, τ_2 *et* τ_3*. En effet, les laxités en cet instant sont* $l_{\tau_1} = 1$, $l_{\tau_2} = 1$ *et* $l_{\tau_3} = 0$*. Ainsi, la tâche* τ_3 *doit commencer son exécution car sa laxité est la plus petite. L'autre processeur peut être occupé indistinctement par* τ_1 *ou* τ_2*. Si on exécute* τ_1 *(figure 1), alors l'ordonnancement sera correct. Par contre, si on choisit d'exécuter* τ_2 *(figure 2),* τ_1 *n'est peut commencer son exécution que à* 1*, avec une laxité* 0*. À l'instant* 2*,* τ_4 *et le deuxième job de* τ_3 *s'activent, en conséquence trois tâches avec une laxité nulle doivent s'exécuter sur deux processeurs. Une des tâches ratera nécessairement son échéance.*

4.3 Génération de paramètres hors-ligne

Dans cette partie, nous nous intéressons à la génération d'ordonnancements corrects par construction. Dans une première partie, nous regardons les choix d'affectation de priorité. En effet, il n'existe pas d'algorithme simple optimal pour choisir les priorités des tâches lors de l'utilisation d'un ordonnanceur FP. Dans une deuxième partie, nous proposons de calculer hors-ligne un ordonnancement valide avec préemption et migration. La méthode est optimale dans le sens où s'il existe un ordonnancement valide, elle le trouvera. Ces deux types d'ordonnancement sont particulièrement utilisés dans l'industrie des systèmes embarqués critiques car leur mise en œuvre est très simple.

4.3.1 Affection hors-ligne de priorité fixe

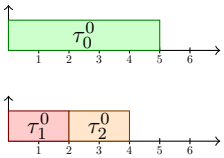
L'affectation sûre et efficace de priorité aux tâches est un problème connu dans le cas monoprocesseur. *Rate Monotonic* (les priorités sont affectées par ordre croissant des périodes) est optimal pour les ensembles de tâches indépendantes synchrones à échéances implicites, *Deadline Monotonic* (les priorités sont affectées par ordre croissant des échéances) est optimal pour les ensembles de tâches indépendantes synchrones à échéances contraintes, et l'algorithme d'Audsley [Aud91] est optimal pour les ensembles de tâches indépendantes asynchrones à échéances contraintes. *Deadline Monotonic* et l'algorithme d'Audsley ont été étendus dans [FBG+10] pour prendre en compte les contraintes de précédence généralisées.

Malheureusement, aucune de ces politiques ne restent optimales dans le cas de plateforme multiprocesseurs.

Exemple 32. *Pour l'ensemble de tâche synchrone à échéance implicite ci-dessous, la tâche τ_0 doit nécessairement avoir une priorité plus élevée que τ_1 ou τ_2. RM échoue donc à ordonnancer cet ensemble de tâche sur 2 processeurs.*

Task set

τ_i	T_i	C_i	O_i	D_i	$prio_i$
τ_0	6	5	0	6	1
τ_1	5	2	0	5	2
τ_2	5	2	0	5	3



Configuration representation

time	*0*	*1*	*2*	...
τ_0	$(6,5,0,6)$	$(5,4,0,5)$	$(4,3,0,4)$	
τ_1	$(5,2,0,5)$	$(4,1,0,4)$	$(3,0,0,3)$	
τ_2	$(5,2,0,5)$	$(4,2,0,4)$	$(3,2,0,3)$	

L'algorithme d'Audsley ne peut pas être translaté dans le cas multiprocesseur. En effet l'idée originale est de tester si une tâche accepte la priorité la plus faible en vérifiant l'ordonnançabilité de l'ensemble de tâche sans affecter les priorités des autres tâches. Un tel test n'existe pas en multiprocesseur et il faut nécessairement affecter toutes les priorités aux tâches. C'est la raison pour laquelle Cucu et Goossens [CG07] ont proposé une solution "brute force" : l'idée est de chercher parmi toutes les affectations possibles une ordonnançable. La complexité pire cas d'un tel algorithme est de $n! \times c$ où n est le nombre de tâches et c est le coût de la vérification de l'ordonnançabilité de l'affectation.

Implantation de l'exploration brute force

Nous avons implanté la solution brute force de Cucu et Goossens. L'algorithme se fait en deux temps :

1. choix d'une affectation des priorités
2. vérification de l'ordonnançabilité avec l'algorithme de parcours exhaustif montré dans la section précédente. Comme nous sommes avec FP, le parcours se fait sur la fenêtre $[0, H]$ pour les tâches synchrones et $[0, S_n + H]$ pour les tâches asynchrones indépendantes [CGG11]. Nous avons rappelé la formule dans l'état de l'art équation 2.5 page 25. Pour les tâches dépendantes, nous utilisons la condition d'arrêt par recherche d'une configuration déjà visitée comme illustré dans la section précédente.

A nouveau nous avons codé l'algorithme en Uppaal et C. Il faut simplement modifier et tester les différentes valeurs de la liste *list_prio* à l'initialisation. Il s'agit d'un problème de calcul des permutations de l'ensemble $[1, n]$.

Version C

On utilise un algorithme de génération de toutes les permutations de $[1, n]$ puis pour chaque permutation on teste si l'ensemble de tâches est ordonnançable avec l'algorithme précédent. Nous

avons utilisé l'algorithme de génération de toutes les permutations proposé dans [Fer11] que nous rappelons dans 4.19. La fonction principale devient alors celle décrite dans 4.20.

Fonction 4.19 : Calcul des permutations Nextprio

```
1  i=TASK_NUMBER-2;
2  tant que i >= 0 ∧ list_prio[i] > list_prio[i + 1] faire
3      i–;
4  si i >= 0 alors
5      j=TASK_NUMBER-1;
6      tant que list_prio[j] < list_prio[i] faire
7          j–;
8      aux=list_prio[i];
9      list_prio[i]=list_prio[j];
10     list_prio[j]=aux;
11     i++;
12     j=TASK_NUMBER-1;
13     tant que i < j faire
14         aux=list_prio[i];
15         list_prio[i]=list_prio[j];
16         list_prio[j]=aux;
17         i++;
18         j–;
```

Fonction 4.20 : Fonction principale main

```
1  sched=false;
2  tant que not sched faire
3      unhappy = 0;
4      Fonction test ordonnançabilité 4.16 ou 4.17;
5      si unhappy=0 alors
6          sched=true;
7      sinon
8          Nextprio () 4.19;
9  si sched alors
10     imprimer("ensemble de tâches ordonnançable")
11 sinon
12     imprimer("ensemble de tâches non ordonnançable")
```

Version Uppaal

Pour la vérification d'ordonnançabilité, nous avons construit un automate qui ne doit jamais sortir de l'état `ok`. Dans le cas de la recherche d'une affection de priorité optimale, il faut produire une sortie (une valeur de `list_prio`) qui soit correcte par construction. En utilisant le model checker, il faut donc chercher à atteindre un état valide quand l'affectation est correcte. L'automate doit donc être modifié.

La figure 4.7 montre ce nouvel automate. Il est composé de trois états *swap*, *search* et *OK*. Si l'état *OK* est atteint alors il existe une affectation de priorité valide pour l'ensemble de tâches. La recherche est donc un problème d'accessibilité et s'exprime avec la formule en logique temporelle suivante :

$$E <> OK$$

Initialement, l'automate est dans l'état *swap* qui est destiné à la permutation de toutes les valeurs de priorité. La valeur initiale de la liste *list_prio* est $[1, \ldots, n]$. La boucle (1) exécute

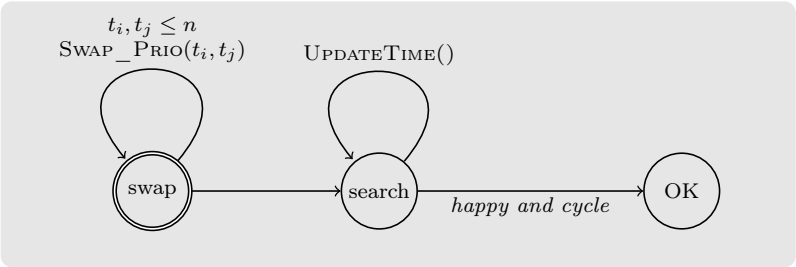


FIGURE 4.7 – Automate de recherche

la fonction *swapPrio* qui permute les priorités des tâches deux à deux. Le code est identique à la fonction *swap_priority()* 4.18 page 68. Cela permet d'explorer l'ensemble des permutations. Pour une affectation de priorités donnée, on atteint l'état *search*. Le système reste dans l'état *search*, en avançant l'exécution, tant que les contraintes temporelles sont valides. Si on tombe soit sur l'hyper-période dans le cas synchrone ou sur un état déjà visité dans le cas asynchrone, cela signifie que l'ensemble est ordonnançable pour l'affection de priorité. L'automate peut passer alors atteindre l'état *OK*. Une fois trouvée une solution, la réponse du model checker est la liste de priorités qui donne une séquence ordonnançable.

Algorithme sous-optimal

La méthode brute force a une mauvaise complexité et rencontre le problème de l'explosion combinatoire, en effet au delà de 15 tâches, si les tâches sont ordonnées dans l'ordre inverse de la solution, le programme ne terminera pas. Nous avons également développé une méthode sous-optimale qui obtient un très bon ratio d'acceptation, ce qui est illustré dans le chapitre 6. Considérons un ensemble de tâches $\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$. L'idée est la suivante :

1. les priorités les plus basses $n, n-1..., n-k+1$ ont été attribuées aux tâches $\mathcal{S}' \subseteq \mathcal{S}$.
2. considérons la tâche $\tau_i \in \mathcal{S} \setminus \mathcal{S}'$ qui accepte la priorité $n-k$, cela signifie que la condition REFUSE décrite dans la définition 13 est fausse.
   (a) On choisit une affectation aléatoire des priorités sur les tâches qui n'ont pas encore temporairement accepté de priorité
   (b) On applique le test d'ordonnançabilité :
      i. Si l'analyse réussit, alors l'affectation est correcte et une solution a été trouvée.
      ii. Sinon, si une des tâches $\tau_k \in \mathcal{S}' \cup \{\tau_i\}$ avec une priorité temporaire excède son échéance, alors l'algorithme backtrack jusqu'à l'étape où τ_k a temporairement accepté sa priorité. On cherche alors une autre tâche pour accepter temporairement cette priorité.
      iii. Sinon, on continue la recherche. On retourne à l'étape 1 et on suppose que les priorités $n, n-1..., n-k$ ont été affectées. On cherche une tâche dans $\mathcal{S} \setminus (\mathcal{S}' \cup \{\tau\})$ qui acceptera temporairement la nouvelle plus basse priorité.

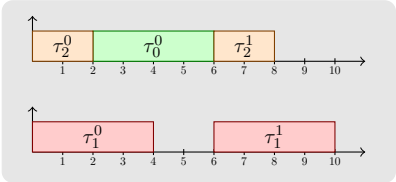
Cette recherche ne visite pas toutes les solutions donc le résultat est soit une solution soit une non-conclusion.

Définition 13 (Condition REFUSE)**.** *Soit l'ensemble de tâche* $\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$. $\tau_1 = (T_1, D_1, C_1, O_1)$ *refuse la plus faible priorité si :*

$$\exists t \leq O_1 + H \ s.t. \ \exists i, \ t - O_i = 0 \ mod \ (T_i), \Sigma_{\tau_i \neq \tau_1} C_i \times \left\lceil \frac{t - O_i}{T_i} \right\rceil + C_1 > t \times m$$

Exemple 33. *Considérons l'ensemble de tâche suivant à ordonnancer sur 2 processeurs :*

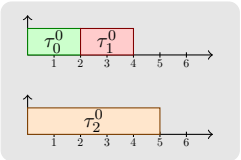
τ_i	O_i	T_i	D_i	C_i
τ_0	0	8	7	4
τ_1	0	6	4	4
τ_2	0	6	4	2



L'heuristique trouve la solution montrée dans la partie droite. En appliquant l'algorithme, τ_0 *accepte temporairement la priorité la plus faible. En effet : pour* $t = 6$, τ_1 *et* τ_2 *s'exécutent exactement une fois. Donc, on a* $(4+2)+4 \leq 6 \times 2$. *Pour* $t = 8$, $(8+4)+4 \leq 8*2$. *Pour* $t = 12$, $(8+4)+8 \leq 12*2$. *Pour* $t = 16$, $(12+6)+8 \leq 16*2$. *Et ainsi de suite jusqu'à* $t = 24$.

Comme seconde illustration, considérons l'ensemble de tâche suivant ordonnançable avec une priorité fixe :

τ_i	O_i	T_i	D_i	C_i
τ_0	0	10	4	2
τ_1	0	10	4	2
τ_2	0	10	5	5



L'heuristique n'arrive pas à conclure. Ce qui prouve qu'elle est sous-optimale. En effet, τ_2 *doit avoir une des deux plus hautes priorités mais aucune des tâches* τ_0 *et* τ_1 *n'accepte la plus faible priorité car elles évaluent la condition REFUSE à vrai.*

4.3.2 Recherche d'une solution optimale hors-ligne

Nous avons présenté dans l'état de l'art plusieurs travaux sur la recherche d'ordonnancement hors-ligne dans le cas d'ensemble de jobs ou d'ensemble de tâches synchrones. C'est le cas notamment de [XP90, XP93] qui utilise une méthode de *séparation et évaluation.*

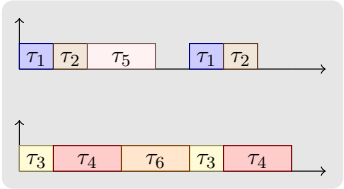
Intérêt d'un ordonnancement hors-ligne

On peut se poser dans un premier temps la question de l'intérêt de rechercher des ordonnancements hors-ligne. Même si aucune politique n'est optimale, on peut se dire que l'ensemble des politiques peut couvrir l'ensemble des systèmes. Ce n'est pas le cas, même pour un ensemble de tâches synchrones. L'ensemble de tâches ci-dessous n'est ordonnançable par aucune des politiques FP (quelque soit l'ordre d'affectation des priorités), gEDF, gLLF et LLREF.

Propriété 5. *L'ensemble de tâches décrit dans le tableau 4.8a est faisable (cf chronogramme d'exécution 4.8b) mais non ordonnançable par une des politiques en-ligne utilisées dans le manuscrit.*

τ_i	O_i	T_i	D_i	C_i
τ_1	0	5	1	1
τ_2	0	5	2	1
τ_3	0	5	1	1
τ_4	0	5	3	2
τ_5	0	8	5	2
τ_6	0	8	7	3

(a) Ensemble de tâches

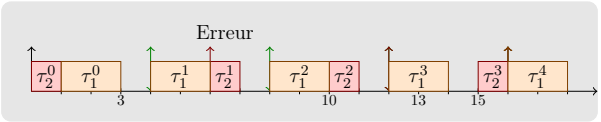


(b) Faisabilité

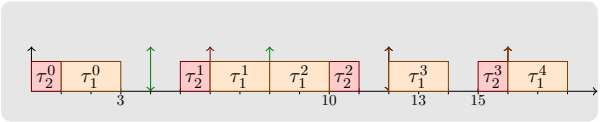
FIGURE 4.8 – Ensemble de tâches faisable mais non ordonnançable par une politique en-ligne

Injection des temps creux

Dans [Gro99], l'auteur souligne la nécessité d'injecter des temps creux pour trouver une séquence optimale valide dans le cas d'utilisation des ressources partagées. Pour deux tâches $\mathcal{S} = \{\tau_1(0,2,4,4), \tau_2(0,1,1,5)\}$ partageant une ressource commune. Il faut injecter un temps creux à 4 pour permettre à τ_2^1 de s'exécuter avant la fin de son échéance comme illustré dans la figure 4.9.



(a) Sans temps creux



(b) Avec temps creux

FIGURE 4.9 – Exécution hors-ligne nécessitant un temps creux

La question est de savoir si ce besoin se transpose également dans le cas de précédence. En fait les temps creux sont inutiles pour les précédences. On introduit deux définitions pour montrer ce résultat.

Définition 14. *L'ensemble des tâches éligibles à l'instant t est l'ensemble de tâches actives et sans contrainte de précédence (ou sans blocage de ressource dans le cas de ressource partagée) à l'instant t après une exécution E entre $[0,t]$. On note cet ensemble $\mathcal{A}_E(t)$.*

Si on reprend les 2 scénarios de la figure 4.9 et que nous calculons les ensembles $\mathcal{A}_{E_i}(t)$. Dans l'exécution 4.9a, on a $\mathcal{A}_{E_1}(4) = \{\tau_1^1\}$ et $\mathcal{A}_{E_1}(5) = \{\tau_1^1\}$. Dans l'exécution 4.9b, on a $\mathcal{A}_{E_2}(4) = \{\tau_1^1\}$ et $\mathcal{A}_{E_2}(5) = \{\tau_1^1, \tau_2^1\}$. Ainsi le fait de commencer la tâche τ_1^1 à 4 va réduire l'ensemble $\mathcal{A}_{E_1}(t)$.

Théorème 2 (séquence optimale sans temps creux)**.** *L'injection de temps creux dans la recherche d'une séquence hors-ligne valide n'est pas nécessaire pour un ensemble de tâche avec des précédences.*

Preuve 5. *On considère une séquence valide avec un temps creux E. On montre qu'il est possible de reconstruire une séquence valide sans temps creux E'. Supposons que le temps creux se produise à l'instant t. Cela signifie qu'un processeur est laissé libre alors que $\tau_i^k \in \mathcal{A}_E(t)$. Soit $\mathcal{A}_E(t+1)$ l'ensemble des tâches éligibles à l'instant $t+1$. On a que $\mathcal{A}_E(t+1) \setminus \tau_i^k \subseteq \mathcal{A}_{E'}(t+1)$ où E' est la séquence telle que τ_i^k s'exécute pendant une unité de temps sur le processeur libre. En effet, dans le cas d'un ensemble de tâches soumises à des contraintes de précédences l'exécution au plus tôt*

d'une tâche ne peut qu'augmenter l'ensemble des tâches éligibles aux instants suivants (hormis elle-même) dans la mesure où son exécution libérera les tâches dépendantes. De plus, l'exécution au plus tôt de la tâche τ_i^k dans E' ne peut pas retarder/bloquer une autre tâche $\tau_i^p \in \mathcal{A}_{E'}(t+1)$ car la politique est préemptive et les tâches ne partagent pas de ressource. La séquence sans temps creux E' exécutera donc toutes les tâches en respectant leurs contraintes (temps réel et précédence) de la même façon que la séquence avec temps creux E. Les seules différences possibles entre E et E' sont les temps creux et le nombre de préemptions. Il est donc toujours possible, pour un ensemble de tâche avec précédence, de construire une séquence sans temps creux à partir d'une séquence avec temps creux. □

Automate Uppaal pour systèmes synchrones

L'automate de parcours, figure 4.11, est composé de deux états *search* et *ok*.

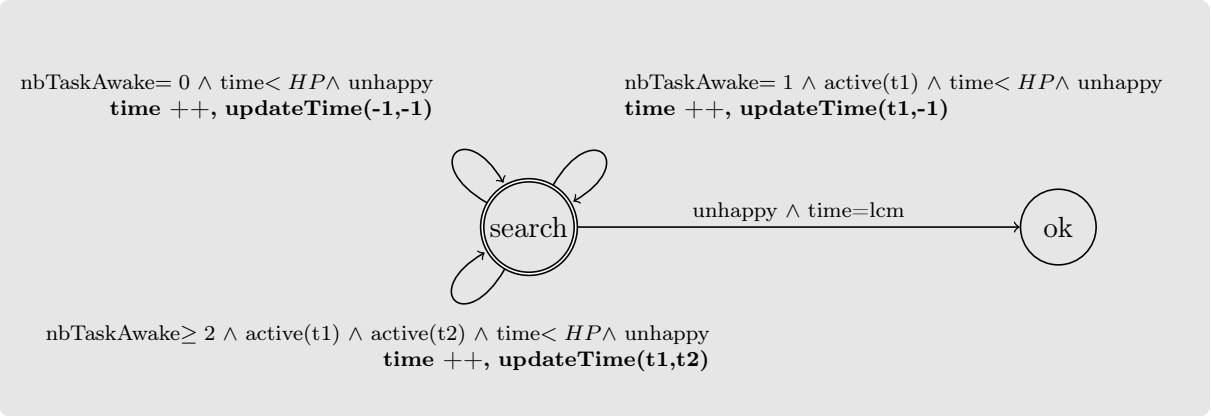


Figure 4.10 – Recherche d'une séquence hors-ligne : cas synchrone pour 2 processeurs

Une variable supplémentaire `nbTaskAwake` compte le nombre de jobs éligibles, cela correspond au cardinal de l'ensemble $\mathcal{A}_E(t)$. Le temps avance jusqu'à l'hyper-période et `unhappy` sert toujours à déterminer si aucune contrainte temporelle n'a été violée. La fonction `updateTime` est légèrement modifiée : elle prend `PROC_NUMBER` entrées. Elle retire -1 au temps d'exécution des numéros de tâches en entrée, c'est-à-dire pour les valeurs strictement positives.

La question posée au model checker est alors de trouver un chemin pour arriver à l'état *ok* :

$$E <> ok$$

Automate Uppaal pour systèmes asynchrones

Le cas asynchrone doit gérer la question de la cyclicité et sauver régulièrement l'état du système. L'automate est représenté dans la figure 4.11 : à nouveau, il y a les deux états *search* et *ok*, les trois transitions temporelles qui mettent à jour les variables des attributs des tâches. Les différences sont :

- utilisation d'une variable supplémentaire `saved` : cette variable est mise à faux à chaque transition temporelle ;
- utilisation des variables `time` et `deb` mise à jour comme illustrée dans l'algorithme 4.4 page 54,

– une transition supplémentaire aux instants *time+deb-MaxO = 0 mod H* qui sauve le vecteur $\langle list_c_var \rangle$ et qui met la variable,
– on peut atteindre l'état `ok` si le dernier vecteur sauvé $\langle list_c_var \rangle$ est égal au vecteur courant (condition `same_vect`).

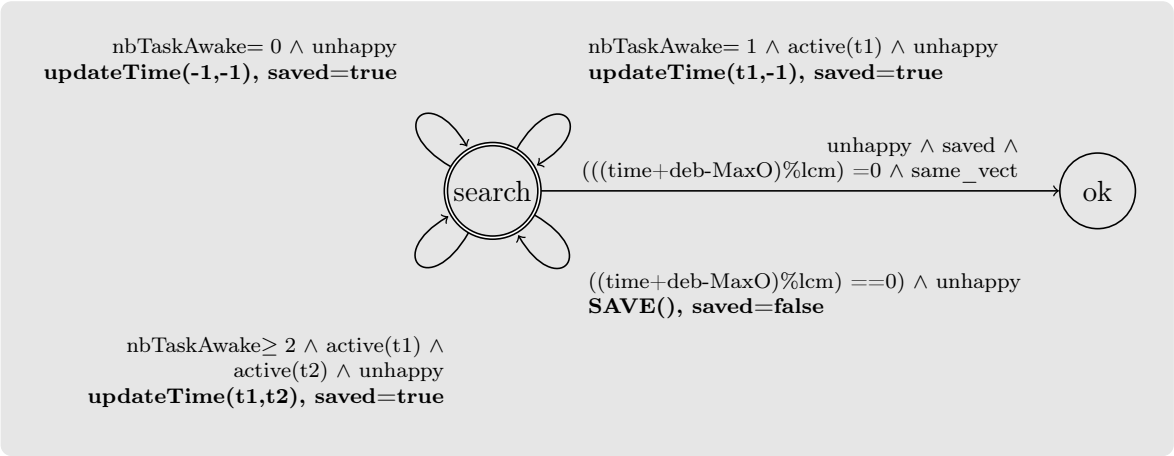


FIGURE 4.11 – Recherche d'une séquence hors-ligne : cas asynchrone pour 2 processeurs

La question posée au model checker reste de trouver un chemin jusqu'à l'état *ok*. La formule logique est la même que dans le cas synchrone.

4.4 Résumé

Nous avons présenté la formalisation et la résolution du problème d'analyse d'ordonnançabilité d'un ensemble de tâches périodiques dépendantes sur des architectures multiprocesseurs. Toutes ces approches ont été outillées dans l'environnement SCHEDMCORE permettant à un utilisateur de décider rapidement de la politique adéquate pour son système.

Chapitre 5

Exécutif SchedMCore Runner

Une fois que nous avons vérifié que le système s'ordonnance théoriquement correctement grâce à SchedMCore Converter, nous désirons l'exécuter sur une cible multicœur. Pour un même ensemble de tâches, on doit pouvoir réaliser des exécutions avec toutes les politiques pour lesquelles cet ensemble est ordonnançable. Dans le chapitre bibliographique 3, nous avons constaté qu'aucune des solutions d'exécution/simulation existantes ne remplissait complètement les conditions nécessaires à nos besoins. C'est la raison pour laquelle nous avons proposé une plateforme d'exécution : SchedMCore Runner. Cet environnement permet d'exécuter un système défini par l'utilisateur conforme au modèle $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$ sur une cible multicœur.

5.1 Architecture de SchedMCore Runner

Dans cette première partie, nous donnons une vue d'ensemble de l'environnement SchedMCore Runner.

5.1.1 Différents niveaux d'utilisation

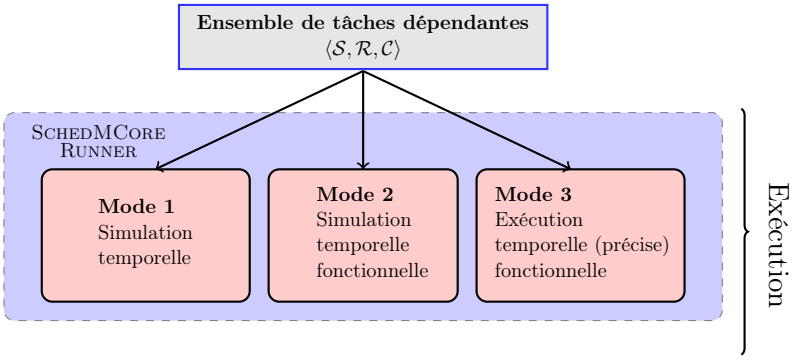


Figure 5.1 – Plateforme d'exécution SchedMCore Runner

L'environnement SchedMCore Runner propose plusieurs niveaux d'exécution. La figure 5.1 en illustre la philosophie : à partir d'un système $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$, il est possible d'exécuter le système selon 3 degrés de finesse. Pour chacun de ces modes on peut choisir la politique d'ordonnancement à utiliser indépendamment de l'ensemble de tâches. Dans le premier mode, *simulation*

temporelle, seul le comportement temporel est simulé. Cela revient à une simulation comme en Storm (cf section 3.2 de la bibliographie). Dans le deuxième mode, la simulation temporelle est identique mais elle inclut l'exécution du code fonctionnel utilisateur. Si la spécification est faite en Prelude, cela revient à une simulation du programme comme la figure 1.3 page 7. Le dernier mode est le plus poussé, car les tâches sont réellement exécutées sur les différents cœurs et le temps est finement calibré. Si ce mode est utilisé sur un OS Linux, le temps réel sera respecté dans les limites des capacités CPU et de la classe de scheduling POSIX FIFO utilisée dans ce cas. L'utilisation d'un RTOS ayant des garanties temporelles meilleures qu'un système Linux n'était pas un objectif de cette thèse. Néanmoins, la conception de SchedMCore Runner doit permettre une utilisation sur un RTOS offrant ainsi des garanties temps réel plus fortes. Il faut remarquer que SchedMCore Runner est conçu pour une architecture multicœur dont la cohérence des différentes mémoires caches est assurée par le matériel de façon transparente pour les applications.

5.1.2 Entités de SchedMCore Runner

La plateforme SchedMCore Runner est composée de plusieurs entités illustrées figure 5.2 : le squelette de l'ordonnanceur, le gestionnaire de temps, les ordonnanceurs et les tâches périodiques génériques. Ces entités sont codées sous forme de threads d'exécution concurrents.

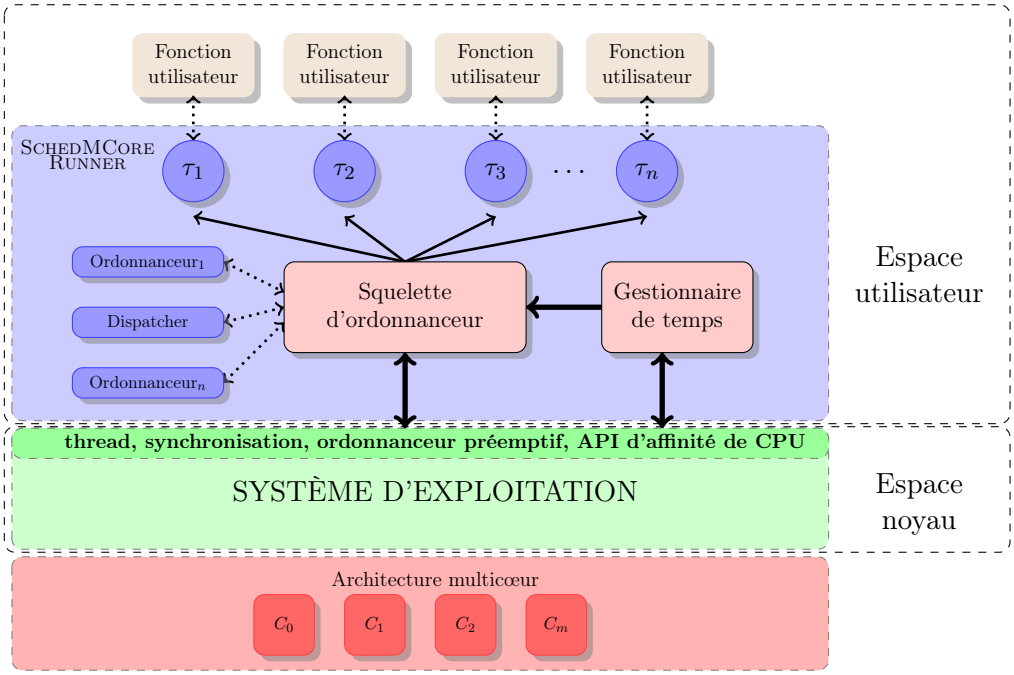


Figure 5.2 – Architecture de SchedMCore Runner

Gestionnaire de temps Le *gestionnaire de temps* est responsable de la gestion temporelle de la plateforme SchedMCore Runner. Pour cela, cette entité communique avec le système d'exploitation et utilise des primitives de lecture d'horloge. Cette entité se réveille à période fixe en utilisant un chronomètre et active l'ordonnanceur.

Dans le futur, il sera possible de gérer plus intelligemment les moments de réveil de l'ordonnanceur. On envisage notamment d'implanter des plateformes dirigées par les événements. Le gestionnaire de temps pourrait donc devenir une *source d'événements d'ordonnancement* parmi d'autres.

Ordonnanceur$_i$ Un *ordonnanceur* est une bibliothèque dynamique (chargée en début d'exécution) qui se greffe ou s'intègre au *squelette d'ordonnanceur*. Cette entité implante une politique d'ordonnancement (comme FP ou gEDF) et doit fournir trois fonctions d'interface :

- `initialize` : cette fonction s'exécutant au démarrage du système est responsable de l'initialisation de l'ordonnanceur, par exemple initialisation de variables locales, affectation des priorités à l'instant initial, etc ;
- `schedule` : cette fonction est responsable de la mise à jour de l'état des tâches, elle est appelée par le squelette à chaque fois que le gestionnaire de temps le réveille ;
- `finalize` : cette fonction s'exécutant à la fin de l'exécution est responsable de la destruction de la mémoire dynamique réservée. Dans un système réel cette fonction ne serait probablement jamais appelée, car un système temps réel critique fonctionne généralement indéfiniment.

Cette séparation entre l'ordonnanceur et le squelette permet deux choses. Premièrement, le chargement d'une politique d'ordonnancement ne nécessite pas de recompiler tout le système. Cette généricité permet d'inclure aussi bien les ordonnanceurs en-ligne que les hors-ligne (dispatcher). Deuxièmement, l'utilisateur peut facilement ajouter de nouvelles politiques d'ordonnancement. Le codage d'une nouvelle politique ne tient pas compte de l'interaction avec le système d'exploitation puisque cette partie est prise en charge dans le squelette. A titre d'illustration, nos implantations de gEDF et FP font moins de 50 lignes de code C.

Squelette d'ordonnanceur Le squelette d'ordonnanceur applique les décisions prises par l'ordonnanceur auquel il est lié en effectuant les appels systèmes nécessaires (comme l'affinité processeur ou le changement de priorité). L'ordonnanceur choisi pour l'exécution, appelé sous forme d'une librairie dynamique, est automatiquement intégré (ou greffé) dans le squelette. Dans la figure 5.3 nous montrons l'entité squelette d'ordonnanceur à laquelle on a greffé la fonction ordonnanceur$_i$ parmi les ordonnanceurs éligibles.

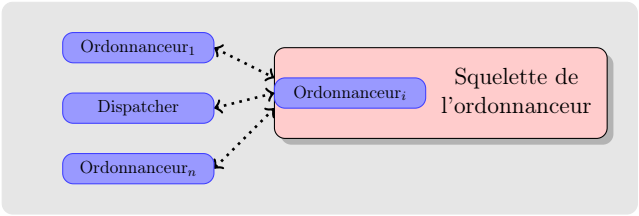


Figure 5.3 – Ordonnanceur$_i$ greffé dans le squelette

Le squelette applique la politique d'ordonnancement, c'est-à-dire l'ensemble des actions à réaliser à cet instant (préemption, réveil d'une tâche, allocation d'une tâche sur un cœur). Pour cela, le squelette interagit avec le système d'exploitation pour pouvoir gérer l'accès des tâches aux cœurs. Une fois les actions réalisées, le squelette se suspend.

Tâche temps réel périodique La dernière entité est celle de la tâche temps réel périodique. En général l'utilisateur ne fournit que la partie fonctionnelle pure de la tâche, i.e. une fonction C. Pour l'exécution effective, chaque tâche décrite est encapsulée dans un thread d'exécution qui appellera le code fonctionnel exprimé en C (si fourni) ou la fonction consommatrice de temps (sinon). Dans la figure 5.4 nous montrons une tâche temps réel périodique à laquelle on a greffé une fonction utilisateur.

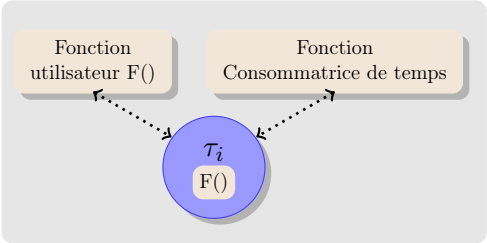


Figure 5.4 – Fonction F() greffée dans une tâche

Interactions entre les entités Ces entités interagissent durant l'exécution d'un système temps réel. Nous décrirons en détail le fonctionnement général de SchedMCore Runner dans la partie 5.2. L'exécution se fait en deux phases : une première phase d'initialisation pour lancer les threads et la gestion du temps, puis une phase nominale d'exécution. Nous avons choisi pour la phase nominale une implantation dirigée par le temps dans le sens où à chaque unité de temps nous appliquons le même traitement.

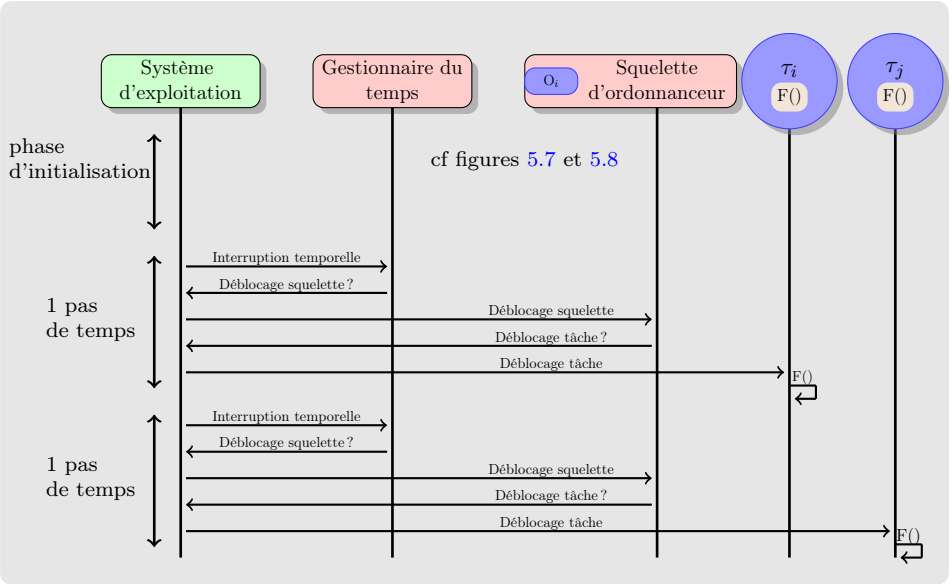


Figure 5.5 – Séquence des interactions entre les entités

Nous décrivons le fonctionnement de l'exécutif dans la figure 5.5. Cette figure montre, sous forme de diagramme de séquence, le fonctionnement dirigé par le temps et les interactions entre les entités et le système d'exploitation. Il y a une première phase d'initialisation que nous décrirons

plus tard. Le fonctionnement nominal démarre ensuite et nous illustrons deux pas de temps. A chaque pas, la séquence suivante se produit :

1. le système d'exploitation réveille le gestionnaire de temps ;
2. le gestionnaire de temps fait une demande de réveil du squelette d'ordonnanceur au système d'exploitation et s'endort ;
3. le système d'exploitation réveille le squelette d'ordonnanceur ;
4. le squelette applique les mises à jour et décide quelle(s) tâche(s) exécuter. Il demande au système d'exploitation de les réveiller et s'endort ;
5. le système d'exploitation réveille la(es) tâche(s) requise(s) ;
6. chaque tâche lance sa fonction `F()` et s'endort.

5.1.3 Dynamicité et portabilité

La plateforme SchedMCore Runner a été développée dans un souci de généricité et de facilité d'utilisation. L'environnement est conçu de façon suffisamment modulaire pour permettre de changer/ajouter facilement la politique d'ordonnancement et/ou le jeu de tâches à étudier. Nous avons donc opté pour le chargement de librairies dynamiques de façon à simplifier l'ajout de nouveaux composants et pour l'utilisation de services compatibles avec le plus grand nombre de systèmes d'exploitation possible.

Gestion des librairies dynamiques

La plateforme SchedMCore Runner a été conçue de façon à assurer la liaison à l'exécution de trois types de librairies dynamiques : les ordonnanceurs, les fonctions utilisateur des tâches et les séquences d'exécution obtenues hors-ligne. La liaison dynamique facilite l'intégration d'un système car on évite de compiler toute la plateforme avec l'application à exécuter à chaque nouvelle utilisation. On peut compiler plusieurs applications et plusieurs ordonnanceurs et les charger au moment de l'exécution.

Lors d'une intégration du système sur une cible embarquée réelle on ne procéderait pas de la sorte. En effet, dès que la plateforme est installée, on ne cherche pas à changer dynamiquement ses composants. Les seules modifications se font durant les mises à jour et à donc à l'arrêt. Notre choix d'utiliser le chargement dynamique est venu de notre objectif de validation du système et non de réaliser l'implantation finale. A noter toutefois, que la conversion pour une solution non dynamique serait relativement simple à mettre en œuvre.

Plusieurs fonctions ont été créées pour faciliter l'utilisation de la liaison dynamique. Pour le chargement dynamique d'un ordonnanceur, on définit :

- Pour charger un ordonnanceur :

  `int lsmc_loadScheduler(libName)`
  - `libName` : nom de la bibliothèque contenant les fonctions attendues pour implanter un scheduler.

- Pour lier la séquence d'un ordonnancement statique :

  `int lsmc_loadSequence(libName)`
  - `libName` : nom de la bibliothèque contenant les données décrivant la séquence du dispatcher.

– Pour lier les fonctions utilisateurs aux tâches temps réels génériques :

```
int lsmc_loadUserFunction(libName, &function)
– libName : nom de la bibliothèque contenant la fonction utilisateur
– function : adresse de la fonction à greffer ;
```

Portabilité

La plateforme SchedMCore Runner est implantée comme une librairie en espace utilisateur devant s'exécuter grâce à un nombre restreint de services exécutifs disponibles sur un système d'exploitation ou un exécutif plus simple. Le système d'exploitation doit fournir plusieurs mécanismes pour supporter l'exécution de SchedMCore Runner :

– un ordonnanceur à priorité fixe préemptif avec au moins cinq niveaux de priorité,
– quelques mécanismes de synchronisation (sémaphores et/ou barrières)
– la capacité d'attacher un thread à un cœur (affinité processeur).

Dans le cadre de cette thèse, SchedMCore Runner a été implanté sur un système d'exploitation Linux mais cette l'implantation pourrait se faire pour *n'importe quel exécutif intégrant les trois mécanismes* ci-dessus. Pour chaque service de SchedMCore Runner, nous avons utilisé des fonctions Posix [27]/Linux standards. Le lecteur peut trouver davantage d'informations sur la mise en œuvre des fonctions Posix dans [Gal95, IEE].

5.2 Implantation de SchedMCore Runner

Dans cette partie, nous précisons l'implantation de l'exécutif.

5.2.1 Implantation multithreadée

Nous avons choisi d'implanter les entités de base de SchedMCore Runner illustrées dans la figure 5.2 par des threads. Un *thread d'exécution* est l'entité d'exécution la plus petite qui peut être ordonnancée par un système d'exploitation. Nous aurions pu choisir de les coder par des processus, mais les threads permettent de partager certaines ressources comme la mémoire ; le changement de contexte est également moins coûteux qu'entre deux processus. La plateforme est composée de $3 + n$ threads d'exécution, (1) le programme principal *main*, (2) le *gestionnaire de temps*, (3) le *squelette d'ordonnanceur* (avec sa politique d'ordonnancement greffée) et les n tâches temps réel (avec la fonction utilisateur/consommatrice de temps greffée).

Création et terminaison Pour la gestion des différents threads, nous avons besoin de :

– une fonction pour la création d'un thread :
– une fonction d'attente de terminaison d'un thread. En effet, le créateur de threads doit attendre la fin des threads créés pour terminer sa propre exécution.

SchedMCore Runner est composé de $n+3$ threads d'exécution dont la phase de création est décrite dans la figure 5.6. Ainsi, le thread `main` est le programme principal d'initialisation de la plateforme. Il est responsable de la création du squelette d'ordonnanceur (avec la fonction ordonnanceur déjà greffée) et des tâches temps réel (avec la fonction utilisateur ou consommatrice de temps déjà greffée). Le gestionnaire de temps est créé par le squelette d'ordonnanceur. De

27. Standard international qui définit l'interface entre les applications et le système d'exploitation, ainsi que la sémantique de chacun des services offerts. Linux respecte (en partie) cette norme.

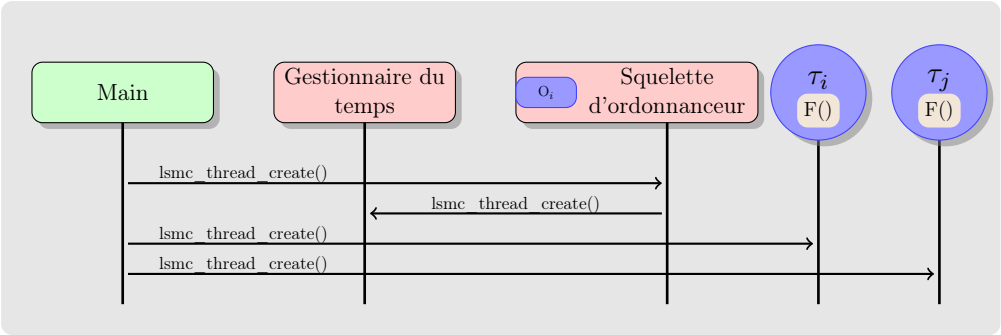


Figure 5.6 – Création de threads dans le système

cette manière le gestionnaire de temps stocke une référence vers son père, ce qui lui permettra de le réveiller.

Préemption Pour gérer la préemption, il faut fournir plusieurs services d'interruption. En effet, les politiques d'ordonnancement implantées dans SchedMCore sont préemptives : à tout moment, l'entité squelette d'ordonnanceur doit être en mesure d'arrêter, de démarrer ou de poursuivre l'exécution d'une tâche sur un cœur. Puisque SchedMCore Runner est une librairie en espace utilisateur, il faut s'appuyer sur l'ordonnanceur du système d'exploitation. Ce dernier doit donc être préemptible également. Pour cela, il faut une fonction de modification de l'ordonnanceur en utilisation.

En outre, pour gérer l'accès simultané aux différents cœurs, nous avons besoin de cinq niveaux de priorités ordonnés par ordre décroissant :

- `LSMC_TIMEKEEPER_PRIORITY` : la priorité la plus élevée est attribuée au gestionnaire de temps. En effet, ce dernier doit pouvoir préempter l'ordonnanceur pour lui indiquer l'écoulement d'un tick d'horloge ;
- `LSMC_MAIN_PRIORITY` : la deuxième priorité est attribuée au programme principal qui démarre tous les threads ;
- `LSMC_SCHEDULER_PRIORITY` : la troisième priorité est attribuée à l'entité ordonnanceur qui doit préempter et libérer les tâches ;
- `LSMC_ACTIVE_TASK_PRIORITY` : la quatrième priorité est attribuée aux tâches en exécution ;
- `LSMC_IDLE_TASK_PRIORITY` : la priorité la plus basse est attribuée aux tâches en attente.

Dans l'implantation Linux actuelle nous utilisons la classe d'ordonnancement Posix SCHED_FIFO qui est préemptible et dispose de suffisamment de niveau de priorité. Un ordonnanceur SCHED_FIFO Posix dispose en général d'au moins 32 niveaux de priorité. Nous avons également besoin d'une fonction permettant la modification de la priorité d'un thread.

5.2.2 Phase d'initialisation

Nous avons déjà expliqué figure 5.5 que l'exécutif se déroule en deux phases : une première phase d'initialisation et une deuxième d'exécution nominale. Contrairement à la phase d'exécution nominale, la phase d'initialisation n'est pas contrainte temporellement. Au moment du démarrage du système, toutes les entités ont une routine d'initialisation à exécuter. Nous avons choisi une initialisation concurrente des entités et deux points de rendez-vous communs réalisés chacun par une *barrière* de synchronisation.

Rappel sur les barrières Une barrière est un mécanisme qui permet de suspendre l'exécution d'un certain nombre de threads pour les activer simultanément une fois la barrière franchie par tous. Pour cela, à la création de la barrière on indique le nombre n de threads à bloquer. Tant que le nombre de threads ayant demandé le passage de la barrière est strictement inférieur à n, tous ces threads restent suspendus. Quand le $n^{ième}$ demande le passage, les n threads sont libérés simultanément. Trois fonctions sont nécessaires pour l'utilisation d'une barrière :

- fonction pour la création d'une barrière ;
- fonction pour la destruction d'une barrière ;
- fonction responsable de demander le passage à une barrière.

Déroulement de la phase d'initialisation La figure 5.7 montre une version schématique du démarrage de SchedMCore Runner, tandis que la figure 5.8 décrit les détails de cette phase. Pour un démarrage simultané de tous les threads, deux barrières sont définies : la première permet l'initialisation de chaque thread. La deuxième permet de démarrer les services utilisés. A noter que le thread `main` n'attend pas la deuxième barrière.

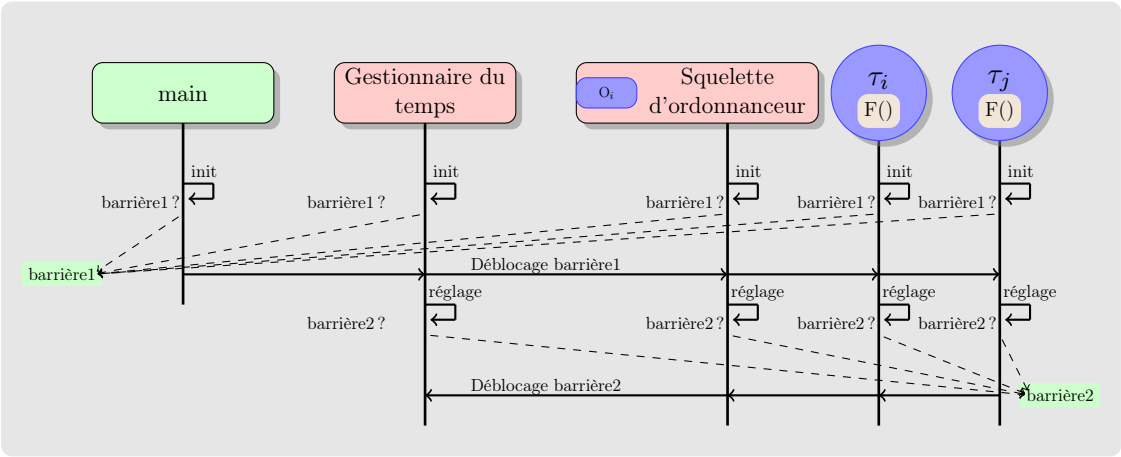


Figure 5.7 – Initialisation simultanée à l'aide de deux barrières

Ces deux figures sont décrites sous forme d'un diagramme de séquence. Les entités se trouvent dans la partie supérieure du diagramme et chacune est associée avec une ligne verticale qui représente l'avancement temporel. Un thread peut agir sur un autre thread, ce qui est représenté par une flèche entre les deux threads concernés. Les fonctions exécutées par chaque tâche sont représentées par une boucle sur la ligne temporelle. Les barrières sont représentées par une ligne horizontale pointillée. Décrivons maintenant en détail l'initialisation de chacune des entités.

Main L'entité *main* est le thread principal du SchedMCore Runner et le premier à être démarré. Dans l'implantation Linux, il s'agit en réalité du processus général mais nous l'assimilons à un thread. La première action à réaliser est la récupération des données temps réel des tâches, $\langle \mathcal{S}, \mathcal{R} \rangle$, afin de les stocker dans des structures internes pour pouvoir les utiliser ultérieurement. La fonction `lsmc_recupererDonnees()` est responsable de cette action. Elle prend les valeurs temps réel et les stocke dans les variables `deadline`, le `releaseDate`, la `period` et le `wcet` de chaque tâche qui se trouvent dans la liste de tâches `taskList[n]`. On peut également introduire les fonctions utilisateur de chaque tâche ou le code naïf de consommation de temps. Pour cela

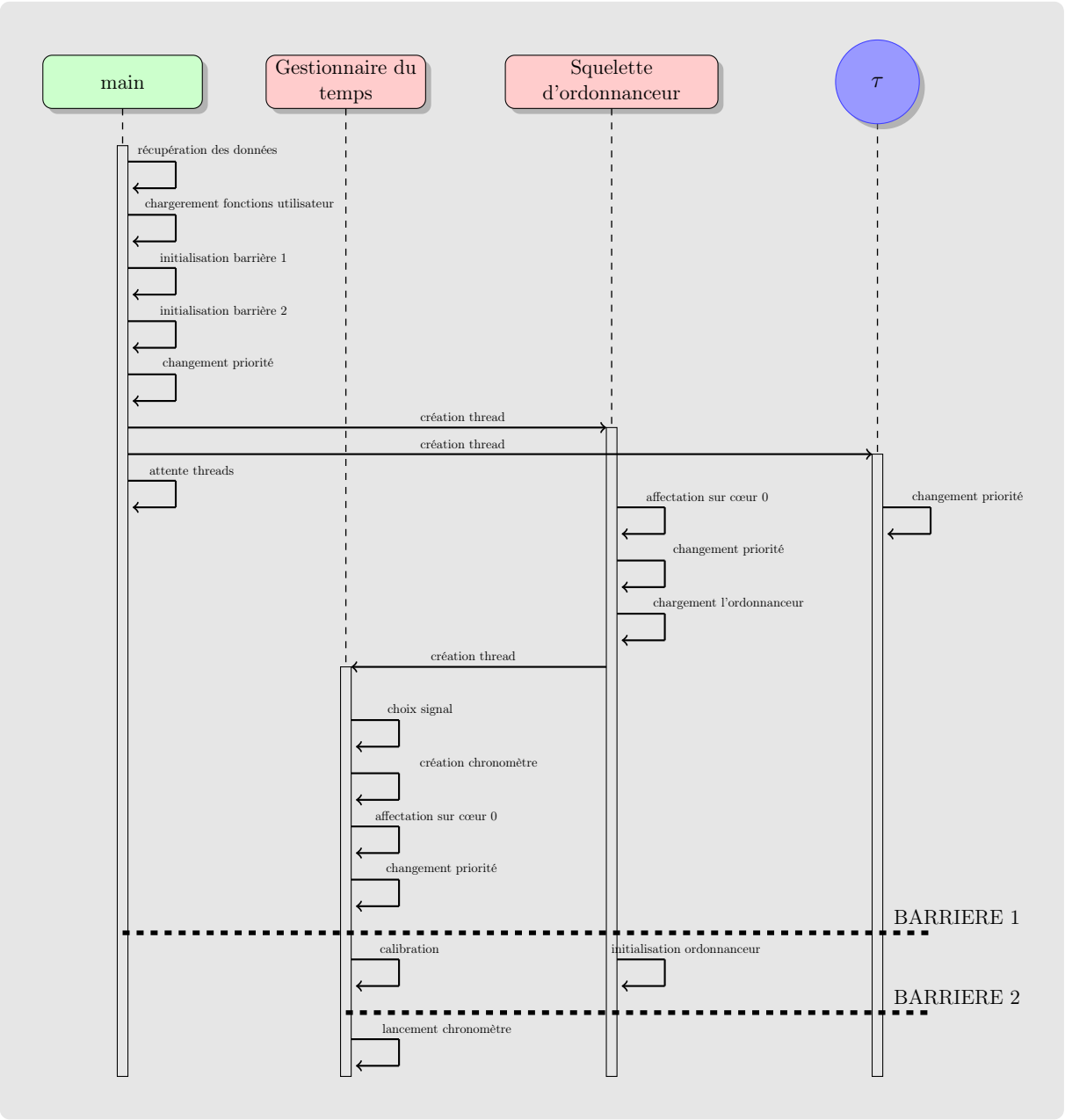


Figure 5.8 – Démarrage de SchedMCore Runner

on utilise la fonction `lsmc_dynamicUserFonctions(&F)`. Cette fonction introduit l'adresse de la fonction `F()` dans la variable `userFunction` de chaque tâche.

On déclare ensuite les deux barrières. La première prend l'identificateur 1 et attend $n + 3$ threads (n est le nombre de tâches temps réel à ordonnancer). La deuxième prend l'identificateur 2 et attend $n + 2$ threads (tous sauf le *main* qui sera en suspension jusqu'à la fin de l'exécution).

Juste après la création des deux barrières, la fonction *main* acquiert la priorité `LSMC_MAIN_PRIORITY` et crée le thread du squelette d'ordonnanceur et les threads des tâches temps réel. La fonction attend ensuite la synchronisation sur la première barrière. Après cette première barrière, la fonc-

tion *main* se suspend (par des appels `lsmc_thread_join`) en attendant la fin de tous les threads qu'il a créés.

Squelette d'ordonnanceur Le squelette d'ordonnanceur est créé par le thread *main*. Au début de son exécution, il change l'affinité de son thread pour s'exécuter sur le cœur 0, et change sa priorité pour acquérir la priorité `LSMC_SCHEDULER_PRIORITY`. Ensuite, il appelle la fonction `lsmc_loadScheduler()` pour charger les fonctions définies dans la bibliothèque dynamique de l'ordonnanceur. Une fois la bibliothèque chargée, le squelette crée le thread du gestionnaire de temps et se suspend sur la première barrière.

Après la première barrière, le squelette exécute la fonction `initialize` (cf description des fonctions génériques des ordonnanceurs section 5.1.2 page 79) de l'ordonnanceur qui a été chargé. Cette fonction varie selon l'ordonnanceur. Dans le cas de gEDF par exemple, la fonction `initialize` 5.1 initialise deux variables : `hp` pour stocker la valeur de l'hyper-période et `instant` pour stocker l'instant courant. Ensuite, les tâches sont ordonnées par l'ordre de priorité dictée par la politique.

Fonction 5.1 : schedEDFInitialize

```
1 H=lsmc_TaskList_hyperperiod(tasksList,nbTasks);
2 instant = 0;
3 sched_edf_order(cthis);
```

Dans le cas du dispatcher, la fonction `initialize` 5.2 commence par l'inclusion dynamique de la séquence. Celle-ci est définie comme un tableau à deux dimensions tel que la ligne a stocke les tâches à exécuter à l'instant a. La valeur de l'`instant` est aussi initialisée à 0.

Fonction 5.2 : schedDispatcherInitialize

```
1 lsmc_link_dynamicSequence(&sequence);
2 dispatcherData->instant = 0;
```

Après avoir initialisé l'ordonnanceur, le squelette se met en attente de la deuxième barrière avant de commencer l'exécution cyclique.

Gestionnaire de temps Le thread gestionnaire de temps est créé par le squelette d'ordonnanceur. Il choisit un signal pour indiquer les interruptions temporelles au squelette d'ordonnanceur avec la fonction `lsmc_setSignal()` et crée ensuite un chronomètre utilisant ce signal. Pour finir, il change son affinité pour s'exécuter sur le cœur 0 et sa priorité pour acquérir la priorité `LSMC_TIMEKEEPER_PRIORITY`.

Après la première barrière, le gestionnaire de temps lance une fonction pour calibrer la vitesse de la machine, pour une plus grande précision temporelle. Après la deuxième barrière, le gestionnaire de temps démarre son chronomètre de façon à avoir une exécution cyclique avec une valeur figée (dans l'exemple 10000).

Tâches Les threads des tâches périodiques sont créés par le processus *main*. Une fois créée, chaque tâche change sa priorité à `LSMC_IDLE_TASK_PRIORITY` ; par défaut, une tâche est toujours initialisée dans cet état. Ensuite, une tâche ne fait que se synchroniser sur les barrières.

5.2.3 Phase d'exécution nominale

Une fois que le démarrage est terminé, le gestionnaire de temps, le squelette d'ordonnanceur et les tâches rentrent dans un comportement cyclique. Le principe a été déjà illustré dans la figure 5.5 page 80. Chaque cycle est déterminé par le chronomètre, défini à l'initialisation par le gestionnaire de temps. Chaque échéance du chronomètre réveille le gestionnaire de temps qui à son tour réveille le squelette d'ordonnanceur. Chacun se suspend ensuite jusqu'à la prochaine période. Le squelette d'ordonnanceur une fois réveillé vérifie s'il doit changer la priorité des tâches temps réel selon la politique choisie. Dans cette section nous expliquons en détail l'implantation de l'exécution cyclique de chaque thread.

Gestionnaire de temps La fonction du gestionnaire de temps est montrée dans l'algorithme 5.3. Elle est composée uniquement d'une boucle avec quatre actions : la première et la dernière instructions mémorisent les temps initial et final de la boucle. Cela nous permet de calculer le temps écoulé entre les deux instants et ainsi avoir un traçage réel et précis de l'exécution. L'instruction de la ligne 3 suspend le thread du gestionnaire de temps jusqu'à l'arrivée du signal `TIMER_SIGNAL_NUMBER`. Ce signal ne peut être émis que par le chronomètre initialisé auparavant dont la période a été définie juste après la deuxième barrière. A son réveil (la réception du signal), il réveille l'ordonnanceur avec l'instruction `lsmc_schedEventRelease (sched)` puis recommence la boucle pour se suspendre sur la ligne 3. A noter que le réveil de l'ordonnanceur se fait naïvement à chaque réveil du gestionnaire de temps. Une fonction pourrait être aisément introduite pour trier plus intelligemment les instants lorsque le réveil est absolument nécessaire et non pas à chaque top du chronomètre.

Fonction 5.3 : lsmcTimeKeeperTask

```
1 tant que 1 faire
2     récupérer temps initial;
3     atteindre signal;
4     réveiller le squelette de l'ordonnanceur;
5     récupérer temps final;
```

Nous utilisons également des mécanismes de synchronisation pour gérer l'accès concurrent des threads aux cœurs. On les utilise, par exemple, pour forcer une tâche inactive à rester suspendue ou pour réveiller une tâche qui devient active. Pour cela, on se sert des mécanismes classiques de synchronisation : les sémaphores d'exclusion mutuelle ou *mutex* et les variables-condition. Les mutex sont utilisés pour protéger l'accès à des ressources partagées : si un thread possède le mutex, aucun autre ne peut utiliser la ressource partagée tant que le thread détenant le mutex ne l'a pas relâché. Les variables-condition permettent à un thread de suspendre son exécution jusqu'à ce qu'une certaine condition (un prédicat) soit vérifiée. Pour utiliser une variable-condition, on lui associe toujours un mutex pour éviter les accès concurrents à cette variable et la protéger. Nous avons développé deux fonctions pour la gestion temporelle entre le gestionnaire de temps et le squelette d'ordonnanceur :

- fonction de réveil de l'ordonnanceur, lancée par le gestionnaire de temps ;
- fonction de suspension de l'ordonnanceur, lancée par l'ordonnanceur.

Tâche temps réel Dans SchedMCore Runner une tâche est une entité qui dispose de deux types de données : celles dédiées au fonctionnement interne de la plateforme et celles dédiées à l'ordonnancement temps réel. Toutes ces données sont stockées dans une structure 5.8. Celles

dédiées au fonctionnement interne sont (1) le pointeur vers la fonction utilisateur `userFunction` et (2) l'identificateur du thread.

Algorithme 5.4: Déclaration des tâches en SchedMCore Runner

```
1  typedef struct Task
2      // données internes
3      lsmc_UserTaskFunction userFunction;
4      // attributs temps réel constants
5      char name[TASK_NAME_MAX_LENGTH];
6      int deadline;
7      int releaseDate;
8      int period;
9      int wcet;
10     // configuration dynamique
11     int job;
12     int executed;
13     int core;
14     stateType task_state;
15     // déclaration des précédences
16     int nbPrec;
17     lsmc_dependency_t* precList;
18 lsmc_task_t
```

Les données dédiées au temps réel sont (1) la chaîne de caractères stockant le nom de la tâche `name`, (2) la `deadline`, (3) le `releaseDate`, (4) la `period` et (5) le `wcet`. Ces valeurs sont chargées au démarrage du système et ne changent pas pendant l'exécution.

Les paramètres `executed`, `job` et `task_state` représentent l'encodage de la configuration de la tâche dans SchedMCore Runner. On retrouve les idées de l'encodage de SchedMCore Converter présenté dans le chapitre 4 :

- `executed` représente le temps d'exécution consommé par le job en cours d'exécution. On a

$$\texttt{executed}(t) = C_i - C_i(t)$$

- `job` correspond au numéro du job en cours d'exécution par rapport à l'hyper-période.

$$\texttt{job}(t) = \left\lfloor \frac{(t - O_i)}{H} \right\rfloor$$

- `task_state` est un type énuméré qui synthétise l'état de la tâche. Les valeurs sont {`execution`, `idle`, `waiting_time`, `waiting_dep`}. La gestion de cette variable dépend du type d'ordonnanceur.

`core` est le numéro du cœur sur lequel le thread est affecté (si dans l'état `execution`).

En dernier, on stocke les contraintes de précédence à respecter par la tâche. Les mots sont décrits dans l'algorithme 5.5. Il s'agit d'un couple d'entiers reliant deux jobs : le job prédécesseur, `jobPrec`, et le job successeur, `jobSucc`.

Algorithme 5.5: Déclaration des mots de précédence

```
1  typedef struct lsmc_precWord
2      int jobPred;
3      int jobSucc;
4  lsmc_precWord_t
```

Chaque contrainte de précédence est déclarée avec le type `lsmc_precedence_t` décrit dans l'algorithme 5.6. On retrouve l'encodage décrit dans la partie 4.1.2 page 54. Une contrainte de

précédence est composée du nom de la tâche successeur, `nameSuccesseur`, du nombre de mots reliant la tâche et la tâche successeur `nbWords` puis la liste des mots stockée dans `word`.

Algorithme 5.6: Déclaration d'une contrainte de précédence

```
1 typedef struct lsmc_dependency
2     char nameSuccessor[TASK_NAME_MAX_LENGTH];
3     int nbWords;
4     lsmc_precWord_t word[NB_DEP_WORD_MAX];
5 lsmc_precedence_t
```

Enfin, la liste des contraintes de précédence associées à une tâche est décrite dans l'algorithme 5.7. La variable `nbPrec` stocke le nombre de contraintes et `listPrec` la liste avec les détails des contraintes.

Algorithme 5.7: Déclaration des contraintes de précédence

```
1 typedef struct lsmc_dependency_list
2     int nbPrec;
3     lsmc_precedence_t listPrec[NB_DEP_MAX];
4 lsmc_precedence_list_t
```

La représentation complète de l'ensemble de tâches se fait par la liste `taskList` contenant les n tâches de type `lsmc_task_t` ainsi qu'une variable `instant` correspondant à l'instant courant. Cet entier est donc actualisé à chaque top d'horloge.

Algorithme 5.8: Déclaration de l'ensemble de tâche en SchedMCore Runner

```
1 typedef lsmc_taskList_t lsmc_task_t taskList[n];
2 int instant;
```

Le comportement d'une tâche périodique est montré dans l'algorithme 5.9. Au début de la boucle, le thread est suspendu sur une variable-condition. Lorsqu'elle est réveillée par l'ordonnanceur, elle stocke le temps initial (ligne 3), et exécute la fonction utilisateur associée si elle est disponible ou appelle la fonction consommatrice de temps sinon. A la sortie de cette condition on stocke le temps. Nous forçons le comportement temporel de la tâche de sorte qu'elle consomme toujours exactement le WCET. C'est la raison pour laquelle, ligne 10, on calcule le temps consommé en réalité et celui restant jusqu'au WCET. On consomme cette différence si elle est supérieure à 0.001s, ligne 12. Finalement, on stocke le temps de la fin le traçage et on finit la boucle.

Fonction 5.9 : PeriodicTask

```
1  tant que 1 faire
2      Atteindre d'être réveillée;
3      récupérer le temps initial;
4      si fonction utilisateur définie alors
5          exécution de la fonction utilisateur;
6      sinon
7          consommer le wcet de la tâche;
8      si le temps initial et le temps courant n'est pas égal au wcet alors
9          Consommer le temps restant;
10     récupérer le temps final;
```

Squelette d'ordonnanceur Le thread de l'ordonnanceur est illustré dans l'algorithme 5.10. La fonction est une simple boucle synchronisée par l'instruction de la ligne 2 où le thread se suspend.

Fonction 5.10 : schedulerSkeleton

```
1 tant que vrai faire
2     lsmc_scheduler_schedEventWait(mySchedulerParams);
3     sched_greffe_schedule(mySchedulerParams);
```

Lorsque le gestionnaire de temps le réveille par réception du signal, le thread exécute la fonction `sched_greffe_schedule(mySchedulerParams)` qui appelle la fonction `schedule` de l'ordonnanceur greffé. Cette fonction est différente selon qu'on utilise un ordonnanceur en-ligne ou un ordonnanceur hors-ligne. Nous montrons le code de la fonction `schedule` pour gEDF (fonction 5.11).

Fonction 5.11 : schedEDFSchedule

```
1 sched_edf_order(taskList);
2 lsmc_updateTasksState(taskList, instant);
3 lsmc_updateCoreState(taskList);
4 instant = (instant+1)%H;
```

La fonction prend en entrée la liste de tâches *taskList* avec les informations temps réel de toutes les tâches. On doit recalculer les priorités des tâches à chaque fois que l'ordonnanceur est appelé et qu'une nouvelle tâche est réveillée. Cette action est réalisée dans la ligne 1. La fonction `lsmc_updateTasksState(taskList,instant)` ligne 2 modifie les variables `task_state` des tâches. Le comportement d'une tâche gérée par un ordonnanceur en-ligne est décrit par l'automate de la figure 5.9.

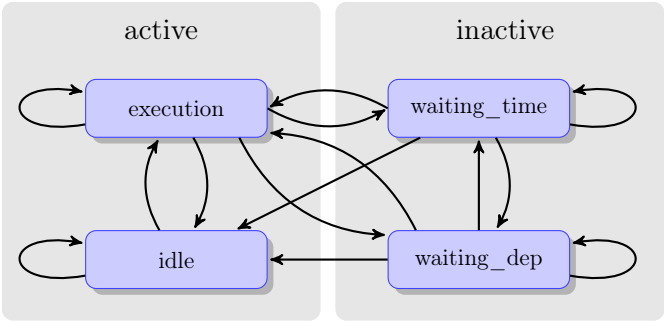


Figure 5.9 – Gestion du paramètre `task_state` par un ordonnanceur en-ligne

Le travail principal de la fonction `lsmc_updateTasksState(taskList,instant)` est de décider à tout moment l'état de chaque tâche, autrement dit, décider à quel moment les transitions de l'automate doivent être tirées. La tâche est dans l'état :

- `execution` quand elle accède à un processeur,
- `idle` quand elle attend un processeur. Les deux états `execution` et `idle` représentent les moments où la tâche est active, c'est-à-dire quand elle est réveillée ($O_i(\texttt{instant}) = 0$), elle n'est pas terminée ($C_i(\texttt{instant}) > 0$) et elle est libre de toute précédence ($libre_i(\texttt{instant})$= $vrai$),
- `waiting_time` quand son exécution est terminée ($C_i(\texttt{instant}) = 0$) et que le prochain job n'est pas activé ($T_i(\texttt{instant}) < T_i$),
- `waiting_dep` quand la tâche est contrainte par une précédence et attend la terminaison d'une autre tâche ($libre_i(\texttt{instant})$=$false$).

Une structure `core` a été définie pour stocker à chaque instant l'état des cœurs. Enfin, ligne 4, on actualise la valeur de l'instant. Si la valeur de l'instant vaut l'hyper-période, on redémarre le compteur `instant` à zéro.

La fonction `dispatcher` pour une exécution hors-ligne se fait à l'aide de la fonction `dispatcherSchedule()` montrée dans l'algorithme 5.12.

Fonction 5.12 : dispatcherSchedule

```
1 pour core ← 0 to nbProc faire
2     getTaskName(instant,core);
3     lsmc_updateTasksState_dispatcher(cthis,instant);
4 pour i ← 0 to TASK_NUMBER faire
5     lsmc_updateTasksState_dispatcher(cthis,instant);
6 instant = (instant+1)%H;
```

La fonction prend également en entrée la liste de tâches *taskList*. On rappelle que la séquence est déjà incluse dans le thread à la fonction initialisation 5.2. La première boucle permet de récupérer dans la séquence du dispatcher (grâce à la fonction `getTaskName(instant,core)`) les tâches à exécuter et de modifier (grâce à la fonction `lsmc_updateTasksState_dispatcher(taskList,instant)`) leur variable `task_state` à `execution`. Le comportement d'une tâche gérée par un ordonnanceur hors-ligne est décrit par l'automate de la figure 5.10. On utilise deux valeurs uniquement pour `task_state`.

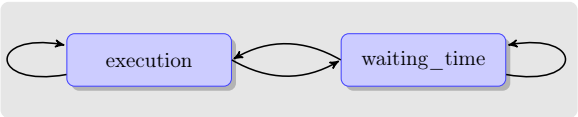


FIGURE 5.10 – Gestion du paramètre `task_state` par un ordonnanceur hors-ligne

La deuxième boucle (ligne 4) met toutes les autres tâches à `waiting_time`. Enfin, ligne 6, on actualise la valeur de l'instant modulo l'hyper-période.

Le changement d'état d'une tâche est géré par les *variable-condition* et les *priorités de l'ordonnanceur sous-jacent utilisé*. Les *variable-condition* sont utilisées pour distinguer les tâches actives (variable-condition bloquée) et les tâches inactives (variable-condition débloquée). Les priorités du système d'exploitation sont utilisées pour différencier les tâches en `execution` ou en `idle`. Au plus m tâches peuvent avoir la priorité `LSMC_ACTIVE_TASK_PRIORITY`.

5.3 Gestion spécifique

5.3.1 Gestion du mode 3

En mode 3, nous cherchons à exécuter le système sur la cible dans des conditions temps réel dures. De ce fait, l'exécutif de SCHEDMCORE RUNNER doit être le plus "léger" possible. Nous avons donc choisi d'exécuter toutes les entités fournisseurs de services sur un unique cœur. Ainsi, le gestionnaire de temps, l'ordonnanceur et le système d'exploitation sont alloués sur un même cœur comme illustré dans la figure 5.11, tandis que les tâches temps réel sont isolées sur un groupe de cœurs qui leur est dédié.

Cette séparation permet de réduire l'effet de l'applicatif sur l'exécution des tâches temps réel. De plus, le support d'exécution peut être non temps réel et les tâches peuvent êtres interrompues

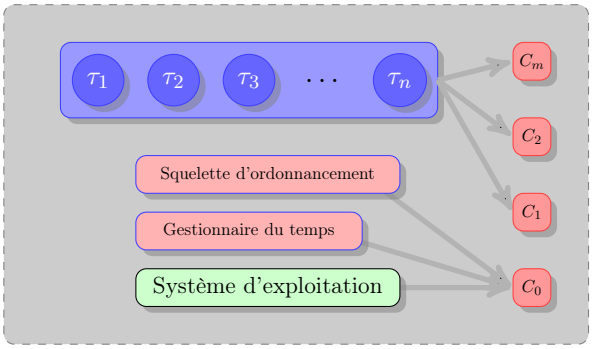


Figure 5.11 – Isolement de l'exécution des tâches temps réel

par le système d'exploitation. Ce choix permet de réduire également ces interruptions non contrôlées. Pour réaliser cette distribution de threads dans les cœurs, une commande d'initialisation du noyau Linux `isolcpus`[28] est disponible, et permet de limiter le nombre de cœurs utilisés par le système d'exploitation.

Gestion du temps

La gestion fine du temps est fondamentale pour le fonctionnement temps réel du mode 3. Deux dispositifs sont nécessaires pour pouvoir maîtriser le temps : les horloges et les chronomètres. Les horloges permettent de compter le temps à partir d'un moment donné. Un chronomètre est un dispositif qui envoie un événement périodiquement. Plusieurs fonctions sont nécessaires pour le gérer :

- fonction de création d'un chronomètre ;
- fonction pour armer un chronomètre.

Les événements lancés par les chronomètres sont des signaux du système. Pour gérer les signaux, on a également besoin de certaines fonctions :

- fonction pour choisir le signal à écouter ;
- fonction pour suspendre un thread jusqu'à l'arrivée d'un signal.

Partage des cœurs Pour gérer la concurrence des tâches temps réel sur les cœurs, on utilise deux fonctions :

- fonction suspension d'une tâche, lancée par la tâche elle-même en fin de cycle (lorsqu'elle n'est plus active) ;
- fonction réveil d'une tâche, lancée par le squelette d'ordonnanceur.

Dans la figure 5.11 nous montrons que les tâches temps réel définies par l'utilisateur s'exécutent sur un ensemble de cœurs. SchedMCore Runner doit alors pouvoir affecter chaque tâche à un cœur précis en tout moment. Cette affectation d'une tâche à un cœur est appelée *affinité de cœur* ou *affinité processeur*. Deux fonctions sont disponibles pour la gestion de l'affinité :

- fonction pour l'affectation d'un thread à un cœur ;
- fonction pour savoir à quel cœur est affecté un thread.

28. http://www.kernel.org/doc/Documentation/kernel-parameters.txt

5.3.2 Besoins spécifiques pour PRELUDE

Gestion des contraintes de précédence par sémaphores Il existe deux approches principales pour gérer les contraintes de précédences. La première solution repose sur l'utilisation de sémaphore : on associe une sémaphore à chaque tâche, bloquant ainsi le job successeur jusqu'à la libération de la sémaphore par le job prédécesseur. La deuxième solution, fonctionnant en monoprocesseur, repose sur l'encodage des contraintes dans les attributs temps comme proposé dans [CSB90, FBLP10] : le job successeur doit se réveiller après le prédécesseur et avoir une priorité plus faible. Nous avons déjà illustré dans la bibliographie page 19 que la transposition de cette méthode en multiprocesseur nécessite de réveiller le successeur après le pire temps de réponse du prédécesseur.

Nous avons choisi dans SCHEDMCORE RUNNER d'utiliser les sémaphores pour gérer par défaut les contraintes de précédence. L'utilisateur peut néanmoins utiliser son encodage hors-ligne avant de fournir l'ensemble de tâches. J. Forget a par exemple coder ses mots d'échéance et de réveil avec SCHEDMCORE RUNNER.

Protocole de communication Le protocole de communication défini dans la thèse de J. Forget pour l'ordonnancement monoprocesseur s'étend directement en multiprocesseur. En effet, les numéros de jobs qui lisent et écrivent sont inchangés en multiprocesseur. Le seul problème pourrait être les moments de disponibilité des buffers, or les buffers sont mis à disposition de la date de réveil du producteur à la fin de la période d'exécution du consommateur. C'est donc lié à l'ensemble de tâches et indépendant du type d'architecture.

5.4 Résumé

Nous avons présenté les choix de conception de l'exécutif SCHEDMCORE RUNNER dédié à l'exécution d'ensemble de tâches périodiques dépendantes avec motifs de communication précis. Dans le prochain chapitre, nous présentons l'environnement SCHEDMCORE, nous donnerons un petit manuel utilisateur de l'exécutif dans la section 6.1 ainsi que des résultats d'expérimentation dans la section 6.3.

Chapitre 6

Expérimentations avec SchedMCore

L'ensemble des contributions est intégré dans l'environnement SchedMCore. Cet environnement contient trois outils : SchedMCore Converter, SchedMCore Runner et SchedMCore Tracer. Le premier, SchedMCore Converter est la mise en œuvre de la méthode d'analyse d'ordonnançabilité pour des politiques en-ligne ou hors-ligne à base de parcours exhaustif décrite dans le chapitre 4. SchedMCore Runner est l'exécutif décrit dans le chapitre 5. Le troisième outil, SchedMCore Tracer, permet le chargement de trace obtenue en Uppaal pour la recherche d'ordonnancement hors-ligne dans l'exécutif SchedMCore Runner.

L'objectif de ce chapitre est d'évaluer les performances de l'utilisation de l'environnement. Concernant l'analyse d'ordonnançabilité, nous souhaitons évaluer si l'approche est applicable sur des ensembles de tâches réalistes. Afin de répondre à cette question, nous prenons en compte comme critères le temps de calcul et le ratio d'acceptation des ensembles, c'est-à-dire le pourcentage de résultats obtenus sur les ensembles de tâches étudiés. Concernant l'exécutif, nous montrons son utilisation sur l'étude du cas du FAS.

6.1 Petit manuel d'utilisation de SchedMCore

SchedMCore est la plateforme qui englobe les applications SchedMCore Converter, SchedMCore Runner et SchedMCore Tracer. Ces applications partagent des structures et méthodes afin de passer facilement de la vérification à l'exécution. Cette plateforme a été écrite en C et la taille du code est montrée dans le tableau suivant :

bibliothèques communes	10551 lignes
SchedMCore Runner	1790 lignes
SchedMCore Converter	15368 lignes
SchedMCore Tracer	1798 lignes
total	29507 lignes

6.1.1 Installation

SchedMCore est téléchargeable à l'url `http://sites.onera.fr/schedmcore/` ou sous forme de svn à l'url `https://svnext.onecert.fr/schedmcore/trunk`. Une fois les sources téléchargées, l'installation se fait aisément en 5 étapes :

1. placez vous dans le dossier *schedmcore*,
2. créez un dossier *build*

3. placez vous dans le dossier *build*
4. tapez *cmake ..*
5. tapez *make*. L'exécutable pour CONVERTER se trouve dans le dossier `build/tool/converter` et pour RUNNER dans le dossier `build/tool/runner`
6. Pour vérifier que les exécutables fonctionnent correctement, tapez dans le dossier `build/tool/converter` : `./lsmc_converter -h` Cette commande doit renvoyer la liste des options du CONVERTER. De la même manière, dans le dossier `build/tool/runner`, tapez la commande `./lsmc_run -h`

Le détail des étapes d'installation est donné dans le fichier `README`. Il faut noter que trois applications sont nécessaires pour la compilation du SchedMCore : `cmake`[29], `bison`[30] et `flex`[31].

6.1.2 Format des fichiers d'entrée

Les deux outils, SchedMCore Converter et SchedMCore Runner prennent les mêmes formats en entrée. Dans la version actuelle de l'outil, l'utilisateur peut décrire le système de tâches sous 3 formes différentes :

1. description textuelle des tâches, c'est-à-dire un système sous la forme de $\langle \mathcal{S}, \mathcal{R} \rangle$ sans code fonctionnel,
2. description de tâches accompagnée du code C de chacune des tâches. On obtient donc un système sous la forme de $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$. On suppose que les variables échangées sont codées par des variables globales.
3. programme Prelude accompagné du code C de chacun des nœuds importés.

Au moment de la conception de cet outil, la flexibilité a été un des facteurs recherchés. Ainsi, le système peut être étendu à d'autres descriptions de systèmes compatibles avec la sémantique $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$.

Un fichier textuel descriptif d'entrée doit respecter un format textuel simple permettant de décrire un ensemble de tâches. Par exemple, pour l'ensemble de tâches $\mathcal{S} = \{\tau_0 = (0, 5, 5, 1), \tau_1 = (0, 5, 5, 1), \tau_2 = (1, 5, 5, 1), \tau_3 = (1, 10, 10, 1), \tau_4 = (1, 10, 10, 1), \tau_5 = (1, 20, 20, 1)\}$ avec les contraintes de précédence $\mathcal{R} = \{(\tau_1, \{(0, 0)\}, \tau_0), (\tau_1, \{(0, 0), (1, 0)\}, \tau_3)\}$ le fichier est :

```
1  # Task "Name" T  C D  O
2  Task   "Tau0" 5  1 5  0
3  Task   "Tau1" 5  1 5  0
4  Task   "Tau2" 5  1 5  1
5  Task   "Tau3" 10 1 10 1
6  Task   "Tau4" 10 1 10 1
7  Task   "Tau5" 20 1 20 1
8  # Dependency "pred" "succs" words
9  Dependency   "Tau1" "Tau0"  0 0
10 Dependency   "Tau1" "Tau3"  0 0 1 0
```

Les commentaires sont exprimés après le signe #. Ainsi les lignes 1 et 8 servent uniquement à donner des informations. La description d'une tâche commence par le mot `Task` suivie du nom de la tâche, la période, le WCET, l'échéance et le décalage initial. Une contrainte de précédence commence par le mot `Dependency` suivi du nom de la tâche prédécesseur, la tâche successeur et les mots qui caractérisent cette contrainte. Le schéma de communication $\mathcal{C}$ n'est pas décrit dans ce fichier.

29. Disponible sur le site http://www.cmake.org/
30. Disponible sur le site https://www.gnu.org/software/bison/
31. Disponible sur le site http://flex.sourceforge.net/

6.1.3 Utilisation SchedMCore Converter

L'outil converter permet de transformer une description d'un ensemble de tâches en un modèle d'analyse d'ordonnançabilité C ou Uppaal. Pour utiliser l'exécutable `lsmc_converter`, il faut jouer avec plusieurs options :

- c : [int] nombre de processeurs ;
- m : [Uppaal |C|all] type de modèle à générer Uppaal, C ou les deux (all) ;
- l : [string] nom du fichier d'entrée, s'il s'agit d'une bibliothèque dynamique Prelude ;
- t : [string] nom du fichier d'entrée, s'il s'agit d'un fichier textuel descriptif ;
- p : [FP|gEDF |gLLF |LLREF |optimalFP|optimal|all] politique d'ordonnancement ;
- d : [determinist|undeterminist] version déterministe ou indéterministe (uniquement pour les politiques qui le nécessite et uniquement en Uppaal).

Analyse d'ordonnançabilité d'une politique en-ligne

L'utilisation de SchedMCore Converter pour une analyse d'ordonnançabilité est résumée dans la figure 6.1.

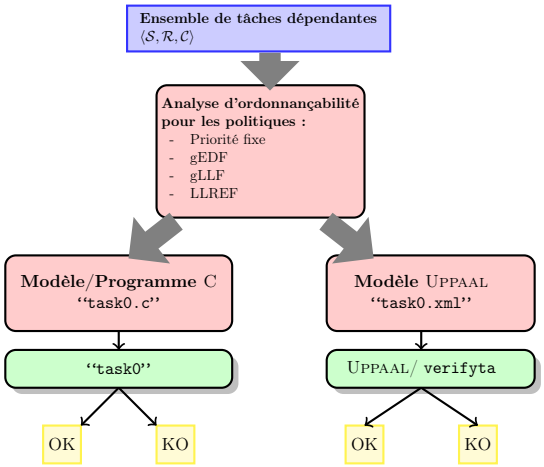


Figure 6.1 – Analyse d'ordonnançabilité en-ligne

Ainsi, pour analyser l'ordonnançabilité en LLREF d'un ensemble de tâches exprimé dans un fichier textuel `task0.txt` sur 2 cœurs avec un modèle C, il faut :

1. générer le fichier modèle C avec la ligne de commande

```
lsmc_converter -t task0.txt -c 2 -m c -p llref
```

2. compiler le fichier généré `task0.txt_LLREF.c`

```
gcc -o task0 task0.txt_LLREF.c
```

   et l'exécuter

```
./task0
```

   Il peut y avoir deux réponses : si l'ensemble de tâches est ordonnançable, la réponse sera

```
CONGRATULATION!! YOUR TASK SET IS SCHEDULABLE WITH LLREF
```

   sinon, la réponse sera

```
I'm sorry!! your task set is NON_SCHED
```

Pour vérifier l'ordonnançabilité du même ensemble de tâches avec Uppaal, il faut :

1. générer le fichier modèle Uppaal avec la ligne de commande

```
lsmc_converter -t task0.txt -c 2 -m u -p llref
```

2. appeler le model checker sur le fichier généré `task0.txt_UppaalLLREF.xml`

```
verifyta task0.txt_UppaalLLREF.xml requete.q
```

   sachant que dans le fichier `requete.q` est stockée la requête

```
A[] not sched_analyser.Loc_Unhappy
```

3. Il y aura également deux réponses : si l'ensemble de tâches est ordonnançable, la réponse sera

```
- Property is satisfied
```

   sinon, la réponse sera

```
- Property is NOT satisfied
```

Génération d'une affectation de priorité

L'utilisation de SchedMCore Converter pour la recherche d'une affectation de priorité est sensiblement identique à l'analyse d'ordonnançabilité, excepté qu'il faut récupérer l'ordre d'affectation. Reprenons l'ensemble de tâche exprimé dans un fichier textuel `task0.txt`. On cherche s'il existe une solution en priorité fixe sur 2 cœurs :

1. pour générer les deux modèles à la fois, il faut taper la commande

```
lsmc_converter -t task0.txt -c 2 -p optimalfp
```

2. pour utiliser le programme C, il faut le compiler et l'exécuter

```
gcc -o task0 task0.txt_OptimalFP.c
./task0
```

   Il y aura alors deux types de réponse, s'il existe une affectation, le message sera

```
CONGRATULATION!! YOUR TASK SET IS SCHEDULABLE WITH OPTFP
list_prio i1, i2, i3 ...
```

   sinon

```
I'm sorry!! your task set is NON_SCHED with OPTFP
```

3. pour le modèle Uppaal, les commandes sont les mêmes que dans le cas précédent et il faut demander la trace du contre exemple.

6.1.4 Utilisation SchedMCore Runner

L'outil runner permet d'exécuter un ensemble de tâches sur une cible multicœur avec une politique d'ordonnancement. Nous avons vu dans le chapitre 5 qu'il y avait 3 modes d'exécution. Les commandes pour l'exécution de ces modes sont :

– *Mode 1* : ce mode correspond à la simulation temporelle du système. Les paramètres temps réel sont stockés dans le fichier `taskFile.txt`. La commande pour exécuter cette simulation est :

```
lsmc_run-nort -t taskFile.txt -s scheduler -c nbProc
```

- *Mode 2* : ce mode réalise une exécution complète du système car on dispose des fonctions utilisateur des tâches dans `userFonctions.so`. Toutefois, cette exécution peut être temporellement non précise car on ne modifie pas l'ordonnanceur du système par un ordonnanceur à priorité fixe préemptif. La commande pour exécuter ce mode est :

```
lsmc_run-nort -l userFonctions.so -s scheduler -c nbProc
```

- *Mode 3* : ce mode réalise une exécution complète du système, comme dans le mode précédent, en revanche, l'exécution est temporellement précise. Ce mode change l'ordonnanceur du système pour utiliser un ordonnanceur à priorité fixe préemptif. Ce mode exige une exécution avec les permissions d'administrateur. La commande pour exécuter ce mode est :

```
sudo lsmc_run -l userFonctions.so -s scheduler -c nbProc
```

Traçage de l'exécution

SchedMCore Runner offre une option de traçage des événements de l'exécution grâce à l'option `-v`. Plusieurs niveaux de verbosité ont été implantés selon le niveau de détail que l'utilisateur désire afficher sur l'écran :

- `-v 0` : ce premier niveau n'affiche aucune information ;
- `-v 1` : ce niveau de verbosité affiche toutes les informations statiques connues au démarrage du système. Ainsi, si le système est constitué d'une tâche Gg dont les paramètres sont $C_{Gg} = 4, P_{Gg} = 40, O_{Gg} = 0, D_{Gg} = 40$, les informations affichées sont :

```
Task 1: name=Gg, deadline 40, release date 0, period 40, priority 6, wcet 4, user↩
    (Function=(nil), Param=(nil), ParamSize=0)
```

  Dans cet exemple, il n'y a pas de fonction utilisateur fournie (donc elle est indiquée avec *nil*) sinon son nom serait précisé.
- `-v 2` : ce niveau montre les informations statiques au moment du démarrage (simplifiées par rapport au *niveau 1*) ainsi que la description de l'évolution des tâches pendant l'exécution. Ainsi, à chaque unité de temps, on affiche pour chaque tâche 6 informations sur la tâche :

  1. `HP` correspond à l'information *est-ce que la tâche a démarré*, on reconnaît la condition $O(\texttt{instant}) == 0$,
  2. `finish` correspond à l'information *est-ce que le job en cours a terminé son exécution*, on reconnaît la condition $O(\texttt{instant}) = 0 \wedge C(\texttt{instant}) = 0 \wedge T(\texttt{instant}) < T$,
  3. `free` correspond à l'information *est-ce que la tâche est libre de contrainte de précédence*, on reconnaît le booléen `libre(instant)`,
  4. `executed` représente le temps d'exécution consommé par le job en cours d'exécution,
  5. `job` correspond au numéro du job en cours d'exécution par rapport à l'hyper-période,
  6. `state` correspond à l'état de la tâche stocké dans la variable `task_state` de type énuméré {`execution`, `idle`, `waiting_time`, `waiting_dep`}.

  Soit un système composé de deux tâches, `tau0` et `tau1`, l'affichage à un certain instant sera :

```
task0: HP:true, finish:false, free:true, executed:0, job:0, state:T_EXECUTION
task1: HP:true, finish:false, free:true, executed:0, job:0, state:T_EXECUTION
```

– `-v 4` : à ce niveau, à chaque instant, on affiche l'allocation des tâches sur les cœurs et l'instant courant. Par exemple, pour un système composé de 2 tâches s'exécutant sur 3 cœurs. On affiche le comportement suivant à l'instant 7 :

```
7 | tau0 | tau1 | X |
```

Cela signifie que `tau0` s'exécute sur le premier cœur, `tau1` sur le deuxième et aucun tâche ne s'exécute sur le troisième.

– Ces différents niveaux sont des masques binaires ($1 = 2^0$, $2 = 2^1$, $4 = 2^2$, ...) que l'on peut cumuler. Par exemple, si l'on demande le niveau `6` on obtiendra l'affichage des informations du niveau 2 et du niveau 4, car $6 = 2^1 + 2^2$. Ainsi, prenons l'exemple d'un système composé de trois tâches, `tau0`, `tau1` et `tau2`, l'affichage sur les 2 premiers instants sera :

```
task0: HP:true, finish:false, free:true, executed:0, job:0, state:T_EXECUTION
task1: HP:true, finish:false, free:true, executed:0, job:0, state:T_EXECUTION
task2: HP:false, finish:false, free:true, executed:0, job:0, state:T_IDLE
 0| task0 | task1 |
task0: HP:true, finish:false, free:true, executed:1, job:0, state:T_EXECUTION
task1: HP:false, finish:true , free:true, executed:1, job:0, state:T_WAIT_TIME
task2: HP:true, finish:false, free:true, executed:0, job:0, state:T_EXECUTION
 1| task0 | task2 |
```

La génération des traces permet à l'utilisateur d'observer et d'analyser finement le comportement du système à l'exécution.

Traitement d'une séquence hors-ligne

SchedMCore Runner peut également exécuter une séquence calculée hors-ligne générée par Uppaal selon la méthode décrite dans la partie 4.3.2. Dans ce cas, on parle de dispatcher plutôt que d'ordonnanceur puisque le dispatcher déroule simplement un échéancier pré-calculé. Chaque ordonnanceur de type *dispatcher* est donc spécifique à un jeux de tâches et au nombre de processeurs sur lequel il doit s'exécuter.

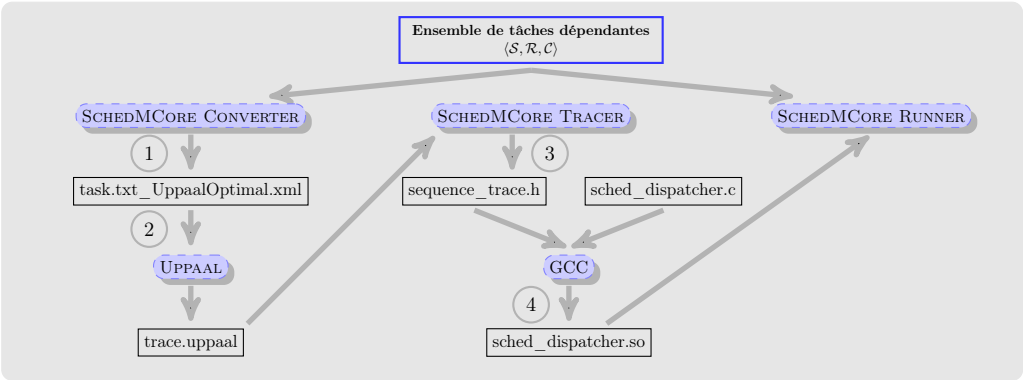


Figure 6.2 – Chargement d'un ordonnancement hors-ligne avec SchedMCore Tracer

La figure 6.2 illustre les différentes étapes pour la construction d'un dispatcher à partir d'un ensemble de tâches :

1. Utiliser SchedMCore Converter pour la politique optimale en demandant la génération d'un modèle Uppaal

   `lsmc_converter -t tasks.txt -c nbProc -p optimal -m uppaal`

Cela créera le modèle Uppaal `tasks.txt_UppaalOptimal.xml`.

2. Récupérer la séquence générée par Uppaal si elle existe. Elle sera fournie sous forme d'un contre-exemple :

   ```
   verifyta -o0 -t0 tasks.txt_UppaalOptimal.xml uppaal_req.q 2> trace.uppaal
   ```

   Ceci créera le fichier `trace.uppaal` qui contient le contre-exemple Uppaal (si il existe). Toutefois, ce fichier est excessivement verbeux (il contient notamment l'information des états explorés) et n'est pas directement utilisable par SchedMCore Runner.

3. Produire un fichier C équivalent à la trace Uppaal compréhensible par SchedMCore Runner. C'est SchedMCore Tracer qui opère cette transformation.

   ```
   lsmc_tracer -f trace.uppaal -t tasks.txt -l c -o sequence_trace.h -c nbProc
   ```

4. Compiler la trace produite `sequence_trace.h` et le patron de code du dispatcher `sched_dispatcher.c` pour produire un ordonnanceur SchedMCore (`sched_dispatcher.so`) chargeable par SchedMCore Runner.

Ces étapes sont un peu fastidieuses mais elles sont adaptées à une utilisation éventuelle sur un système réel pour lequel, in fine, on pourrait utiliser la séquence d'ordonnancement directement dans le code embarqué.

6.2 Expérimentations avec SchedMCore Converter

Pour évaluer les performances de SchedMCore Converter, nous avons mis en œuvre un code générant des ensembles de tâches en fonction d'un ensemble de paramètres et un générateur d'aléas. Ces ensembles de tâches sont stockés sous forme de fichiers textuels descriptifs respectant la syntaxe décrite précédemment section 6.1.2. Nous avons également sélectionné des critères de performance. Puis nous avons lancé de nombreux tests automatiques pour mesurer les performances de SchedMCore Converter.

6.2.1 Génération de tâches

La génération d'ensembles de tâches "pertinents" est un problème complexe car il faut identifier les ensembles de tâches "difficiles à ordonnancer". Cette identification est difficile pour deux raisons :

1. optimalité inconnue : pour mesurer la dureté d'un ensemble de tâches, une méthodologie consiste à chercher des ensembles de tâches faisables mais non ordonnançables par certaines politiques. Malheureusement, en ordonnancement multiprocesseur, la seule méthode actuelle pour générer des ensembles faisables de tâches asynchrones ou à échéances contraintes ou dépendantes est celle du parcours exhaustif qui rencontre rapidement le problème de l'explosion combinatoire ;
2. facteurs inconnus : on ne connaît pas les facteurs déterminants qui interviennent dans l'ordonnançabilité pour une politique donnée. Le taux d'utilisation du système semble *a priori* un facteur crucial même s'il peut y avoir des exceptions comme les anomalies (dans la section 2.3.1) ou l'effet de Dhall [DL78].

Les méthodes pour la génération automatique des ensembles de tâches peuvent donc avoir un effet biaisé et ne pas parvenir à générer des ensembles de tâches représentatifs. Bini et Butazzo [BB04] ont étudié ce problème et proposé une méthode de génération en utilisant une distribution uniforme par rapport à la charge processeur. De cette façon, les auteurs obtiennent des ensembles de tâches avec des duretés diverses. La méthode proposée est montrée dans l'algorithme 6.1.

Algorithme 6.1: UniFast(nbTasks,U)

```
1 U=sumU;
2 pour i ← 1 a nbTasks-1 faire
3     nextSumU=sumU*pow(rdm,(1.0/(nbTasks-i)));
4     vectU[i-1]=sumU - nextSumU;
5     sumU = nextSumU;
6 vectU[nbTasks-1]=sumU;
```

Cet algorithme prend en entrée le nombre de tâches à générer et la charge processeur. En sortie, il génère un vecteur `vectU` qui contient le facteur d'utilisation alloué à chaque tâche. Pour chaque tâche, on génère sa charge et on actualise l'occupation restante `sumU`, pour le reste des tâches. Le SchedMCore Converter repose sur une adaptation de cette technique de génération :

- nous travaillons avec des valeurs entières au lieu de valeurs réelles ;
- nous limitons la taille de l'hyper-période pour éviter une explosion combinatoire. Cette modification est cependant réaliste car les systèmes multipériodiques temps réel se composent, en pratique, de tâches avec seulement un nombre restreint de périodes différentes et/ou avec une relation géométrique entre elles.
- les modifications précédentes empêchent la création de distributions uniformes. En revanche, elles guident la génération pour prévenir la production de tâches trivialement non ordonnançables.

En plus de ces modifications, nous utilisons plusieurs paramètres pour décrire les caractéristiques générales de l'ensemble à générer. Les paramètres de génération sont :

- k : le nombre de tâches à générer ;
- l : la période maximale ;
- q : le nombre de contraintes de précédence à générer ;
- u : l'occupation processeur maximale ;
- r : force la génération des périodes géométriques ;
- s : force la génération d'ensembles de tâches synchrones ($O = 0$).

A partir de la charge de chaque tâche et des paramètres de génération précédents on calcule les paramètres temps réel restants. Les règles de génération sont les suivantes :

$$\begin{aligned} T_i &= l * (1 + (\text{rand}(0, r)) \\ C_i &= vectU[i] * T_i \\ O_i &= \text{rand}(0, T_i) \\ D_i &= \text{rand}(C_i, T_i) \end{aligned}$$

où $\text{rand}(a, b)$ retourne une valeur aléatoire entière de l'intervalle $[a, b]$. Sous Linux le générateur de nombres aléatoires est basé sur la lecture de `/dev/urandom`.

6.2.2 Critères d'évaluation des mesures

Les expérimentations ont été réalisées pour explorer deux facteurs importants : le *taux d'ordonnançabilité* et les *performances temporelles*.

Taux d'ordonnançabilité Un des facteurs primordial pour l'analyse de l'ordonnançabilité est le *ratio d'acceptation* des politiques d'ordonnancement. Celui-ci se définit comme le pourcentage

des ensembles de tâches qui sont ordonnançables avec une politique donnée. A noter comme nous l'avons déjà fait remarqué que, comme il n'existe pas de résultat d'optimalité pour l'ordonnancement multiprocesseurs, on peut difficilement comparer les politiques d'ordonnancement par rapport à une solution optimale, d'où l'utilisation du ratio d'acceptation. Le résultat de la vérification de chaque modèle peut donner lieu à quatre réponses :

1. *OK* : l'ensemble de tâches est ordonnançable avec la politique choisie ;
2. *KO* : l'ensemble de tâches n'est pas ordonnançable avec la politique choisie ;
3. *Time-out* : la vérification a dépassé un temps maximal fixe (pour tous les graphiques présentés dans la suite, ce temps est de 10 minutes) ;
4. *overflow* : la codification d'entiers en Uppaal se fait sur 16 bits. L'hyper-période de certains modèles dépasse cette borne et le modèle ne peut être étudié.

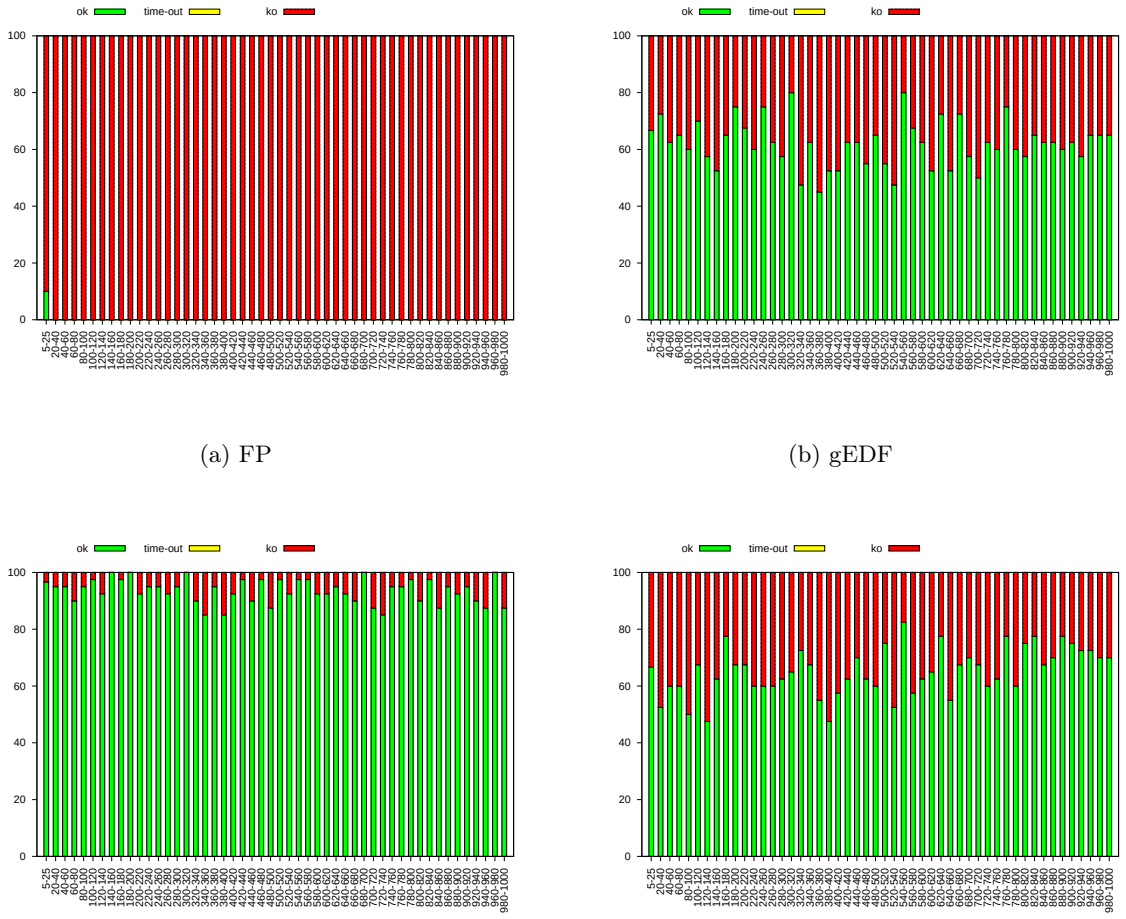
Performances temporelles Nous avons déjà abordé le problème de l'explosion combinatoire des recherches exhaustives. Dans cette section nous présentons quelques résultats empiriques qui vont nous permettre d'évaluer les limites des méthodes utilisées et les comparer. On comparera la durée de vérification des différentes méthodes avant obtention d'une réponse.

Types d'expérimentation Les expérimentations ont donc été menées en faisant varier les paramètres suivants : le *nombre de tâches*, le *nombre de processeurs*, la *présence des contraintes de précédence* et l'*hyper-période*. Ces expérimentations ont été exécutées sur une machine fonctionnant sous Debian Linux (64 bits) équipée de 2 processeurs Intel Xeon X5472 @3.00GHz et de 16Go de mémoire DDR2. Les modèles vérifiés sont en C et Uppaal. La version de Uppaal/`verifyta` utilisée pour la vérification est uppaal v4.1.4 (version 64bits) distribution binaire. Comme dans le cas précèdent, le temps de calcul est limité à une minute pour toutes les vérifications.

6.2.3 Performances de l'analyse d'ordonnançabilité des politiques en-ligne

Nous allons faire varier les paramètres et évaluer les performances des différentes politiques selon les 2 critères de mesure.

Nombre de tâches Un premier facteur déterminant pour l'analyse des politiques d'ordonnancement est la taille de l'ensemble de tâches. Pour analyser son impact, nous générons des ensembles dont le nombre de tâches varie de 5 à 1000 (par pas de 5) et on étudie 10 ensembles de chaque taille. La vérification est lancée pour des ensembles de tâches asynchrones à échéances contraintes indépendantes, une charge processeur de 80%, une hyper-période de 10000, les périodes sont géométriques et il y a au plus 4 périodes différentes. L'architecture est composée de deux processeurs.



(a) FP

(b) gEDF

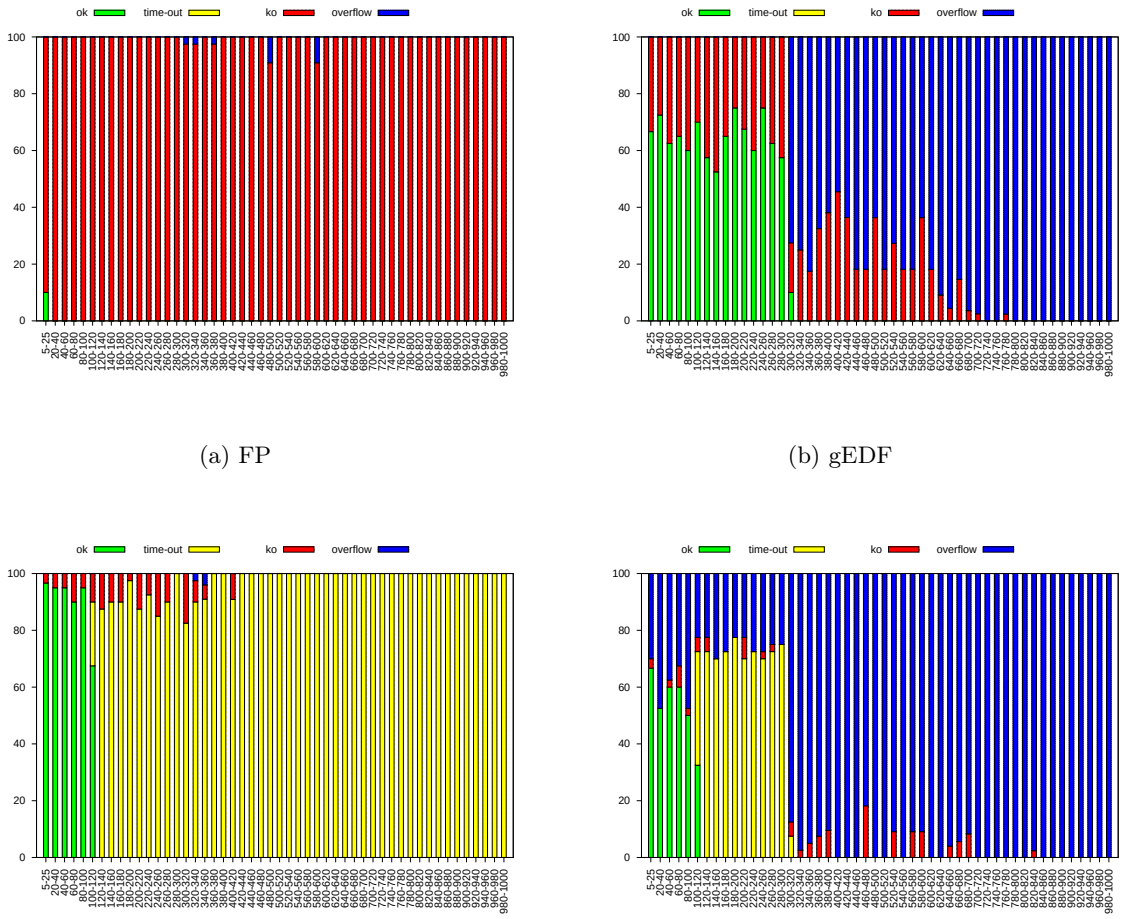
(c) gLLF

(d) LLREF

FIGURE 6.3 – Ratio d'acceptation en C avec variation du nombre de tâches

Les résultats, dans les graphiques 6.3, montrent que la politique FP, dans la figure 6.3(a) n'arrive à ordonnancer que 2 ensembles de tâches. Cela est du au fait que la charge du processeur est élevée comme le confirmera la série de tests sur la *charge du processeur*. La politique gEDF, présentée au graphique 6.3(b), a un ratio d'acceptation moyen d'environ 60%, et ce indépendamment du nombre de tâches. gLLF, présentée au graphique 6.3(c), est la politique la plus performante, avec des taux proches de 90% pour tous les nombres de tâches. LLREF, présenté au graphique 6.3(d) a des résultats proches du gEDF mais le temps de calcul est notablement

supérieur.



(a) FP

(b) gEDF

(c) gLLF

(d) LLREF

Figure 6.4 – Ratio d'acceptation en Uppaal avec variation du nombre de tâches

Les résultats pour les mêmes expérimentations vérifiées cette fois-ci avec les modèles Uppaal correspondants sont présentés à la figure 6.4. Les résultats pour FP en 6.4(a) sont concordants avec ceux obtenus en C. Pour les autres méthodes, on constate que la vérification en Uppaal est nettement moins performante qu'en C puisqu'à partir de 100 tâches pour gLLF et LLREF, et à partir de 300 pour gEDF, la vérification ne termine quasiment jamais. Les ensembles vérifiables à partir de ces limites sont, dans la plupart des cas, des ensembles non ordonnançables avec une vérification assez triviale (le temps de vérification est alors franchement plus petit que celui des ensembles ordonnançables). On peut également noter que avec gLLF et LLREF, à partir de 300 tâches on arrive dans la plupart de cas à la limite de 16bits Uppaal.

La figure 6.5 présente la vue selon le temps d'analyse pour les mêmes ensembles de tâches.

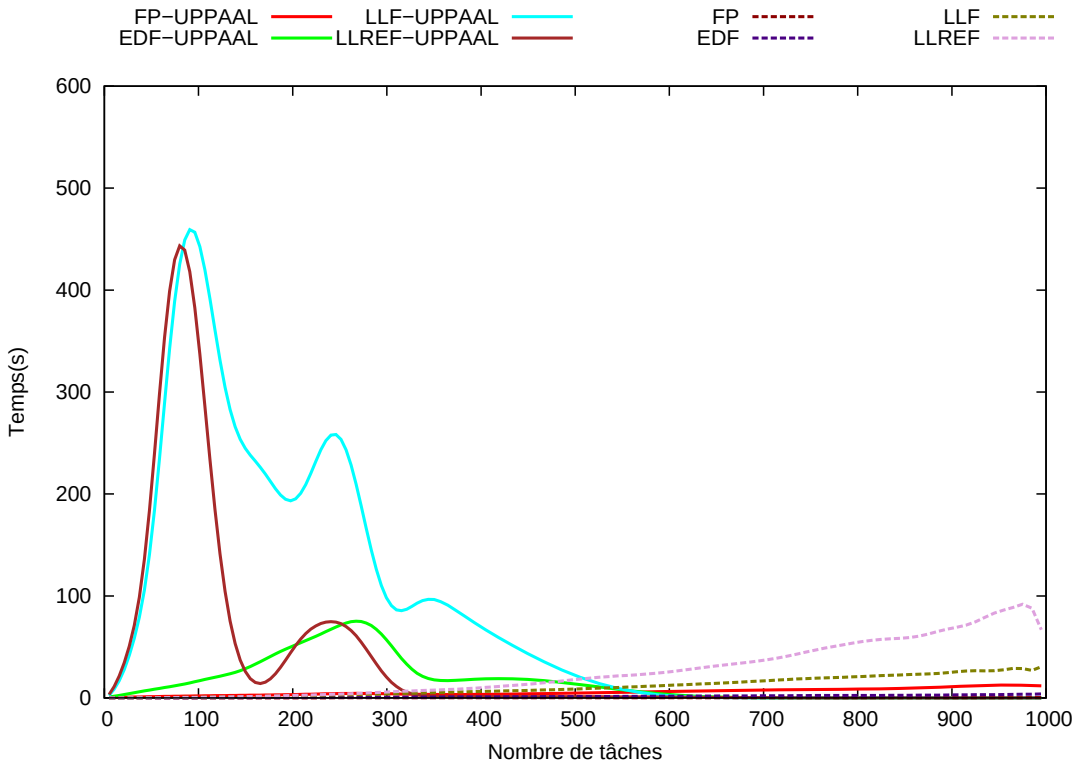
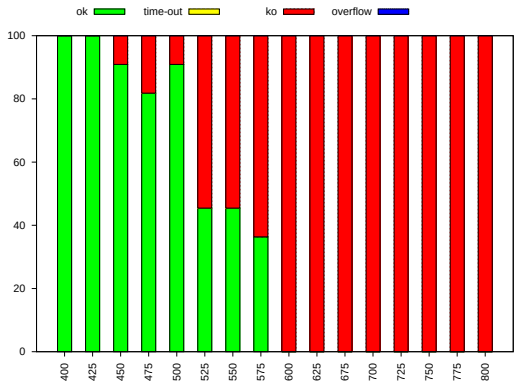


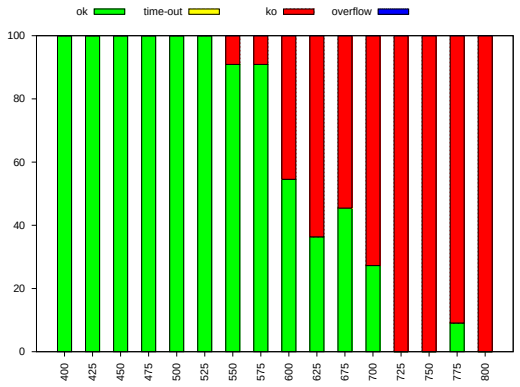
Figure 6.5 – Temps de vérification avec variation du nombre de tâches

Puisque les ensembles ne sont pas ordonnançables avec FP la courbe correspondante ne croit pas au même rythme que les autres dans le cas d'Uppaal. Les modèles Uppaal quant à eux arrivent plus rapidement à une explosion temporelle. gLLF et LLREF s'approchent du temps maximal (10 min) pour la vérification à partir de 115 tâches. Quant à gEDF il parvient à ordonnancer des ensembles de plus de 500 tâches. Les petites fluctuations correspondent à des ensembles non ordonnançables donc le temps de vérification est beaucoup plus petit que 10m.

Charge du processeur Un deuxième facteur important est la charge processeur. L'objectif de ce test est de déterminer le seuil de charge du processeur à partir duquel le ratio d'acceptation d'une politique diminue. Le test a été réalisé sur des ensembles composés de 50 tâches asynchrones à échéances contraintes indépendantes, une hyper-période de 10000, les périodes sont géométriques et il y a au plus 4 périodes différentes. L'architecture est composée de 8 processeurs. La charge processeur varie de 400% à 800% par pas de 25%.

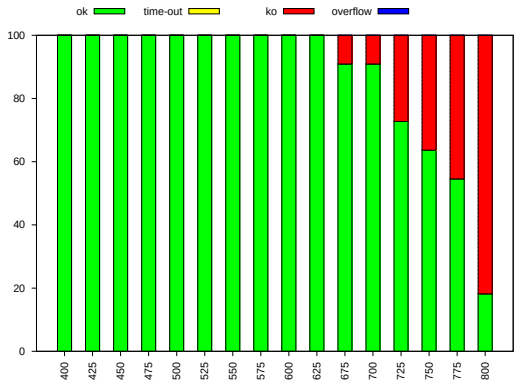


(a) FP



(b) gEDF

(c) gLLF



(d) LLREF

Figure 6.6 – Ratio d'acceptation avec variation de la charge processeur

Les graphiques 6.6 illustrent les ratios d'acceptation obtenus avec les programmes C. Pour ces jeux de test, FP a un bon ratio d'acceptation jusqu'à 500% et au delà de 600% échoue constamment pour ordonnancer les systèmes. gEDF a un bon ratio jusqu'à 600% mais à partir de 725% n'ordonnance presque plus de systèmes. gLLF a un très bon ratio d'acceptation. Dans l'ensemble de nos tests, gLLF est la « meilleure politique d'ordonnancement » dans le sens où elle a toujours les ratios d'acceptation les plus forts. LLREF a également un bon ratio d'acceptation. Uppaal ne conclut pas toujours car 10 minutes sont insuffisantes.

Les performances temporelles du calcul par rapport à la charge du processeur sont présentées

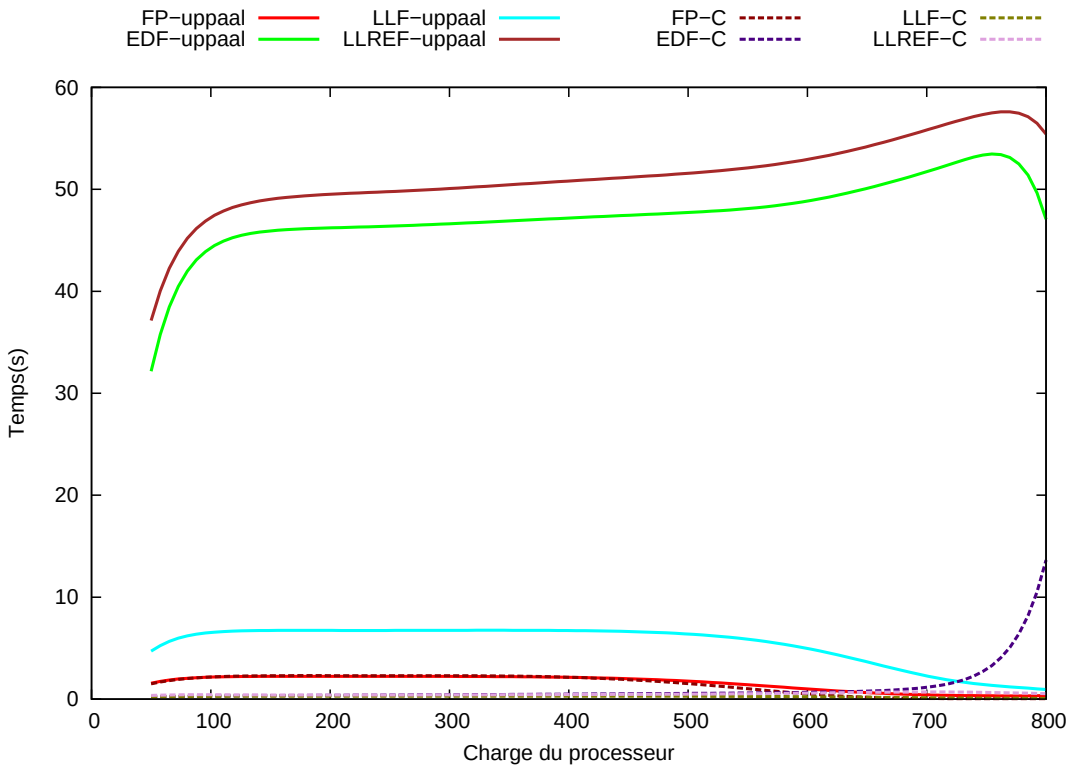
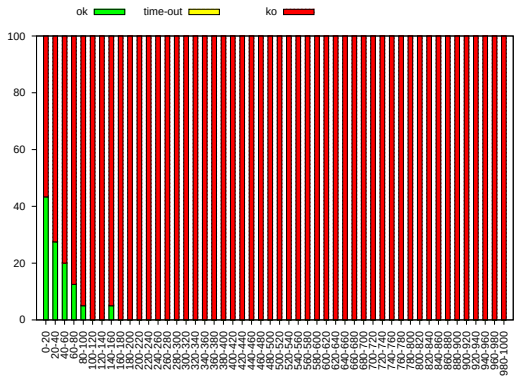


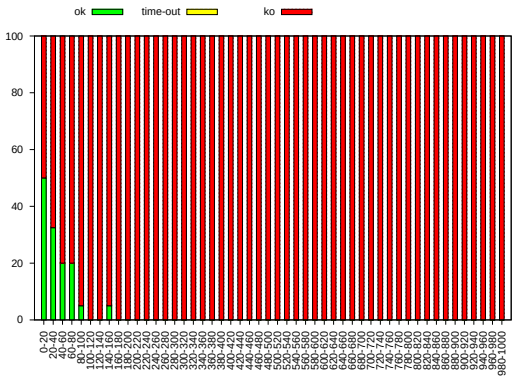
Figure 6.7 – Temps de vérification avec variation de la charge processeur

au graphique 6.7. Au niveau temporel, l'impact de la charge de processeurs est minime. On peut observer l'augmentation de quelques secondes pour les ensembles les plus chargés, par exemple avec le modèle C de gLLF. Également, on trouve la descente de quelques secondes de calcul, qu'on peut interpréter comme un effet biaisé dû à l'apparition d'un grand nombre d'ensembles non ordonnançables.

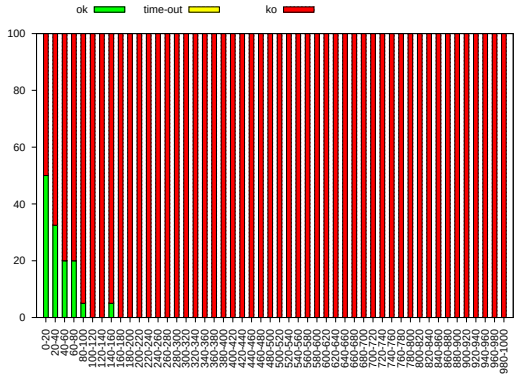
Contraintes de précédence Un troisième facteur influençant les performances est la présence de contraintes de précédence. Le test a été réalisé sur des ensembles de tâches asynchrones à échéances contraintes dépendantes, une charge processeur de 50%, une hyper-période de 10000, les périodes sont géométriques et il y a au plus 4 périodes différentes. L'architecture est composée de 2 processeurs. On fait varier le nombre de tâches (entre 5 et 1000, pas de 5) ainsi que le nombre de contraintes de précédences, on ajoute une contrainte toutes les 20 tâches. Les précédences réduisant le ratio d'ordonnançabilité à nombre de tâches constant, il est nécessaire d'agir sur les 2 paramètres en parallèle.



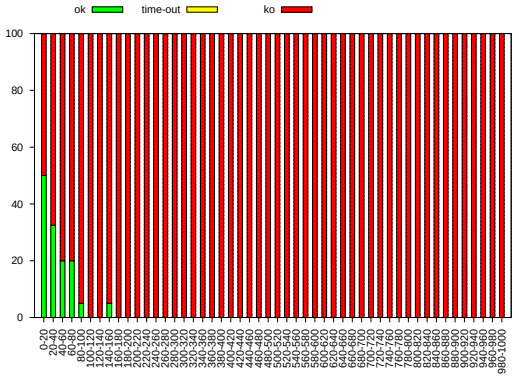
(a) FP



(b) gEDF



(c) gLLF

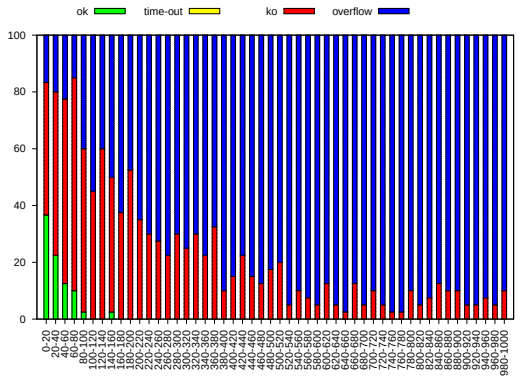


(d) LLREF

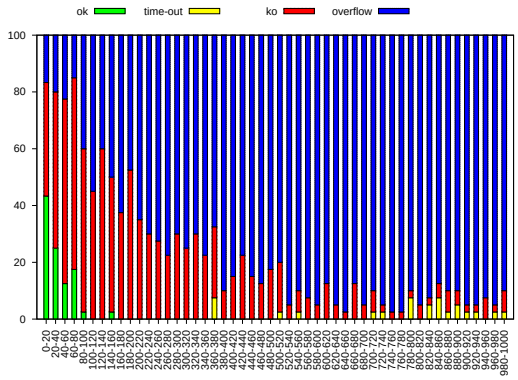
Figure 6.8 – Ratio d'acceptation en C avec variation du nombre de tâche et de précédences

Les résultats sont montrés dans les graphiques 6.8. On peut noter que les résultats pour les quatre politiques d'ordonnancement sont identiques. Si on regarde les ensembles ordonnançables, sont exactement les mêmes. Cette égalité des résultats nous fait supposer que les modèles non ordonnancés sont en réalité non faisables mais nous pouvons pas le garantir.

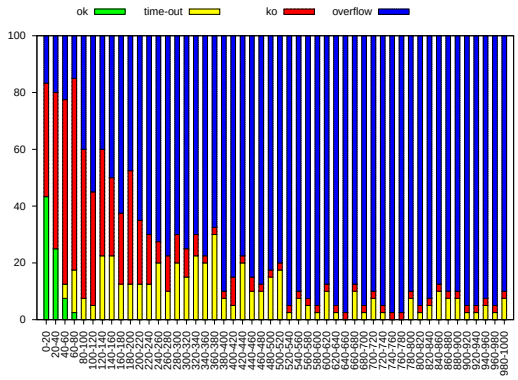
Pour évaluer les performances du modèle Uppaal, nous avons réalisé exactement les mêmes expériences avec les mêmes ensembles de tâches avec Uppaal. Les résultats obtenus sont présen-



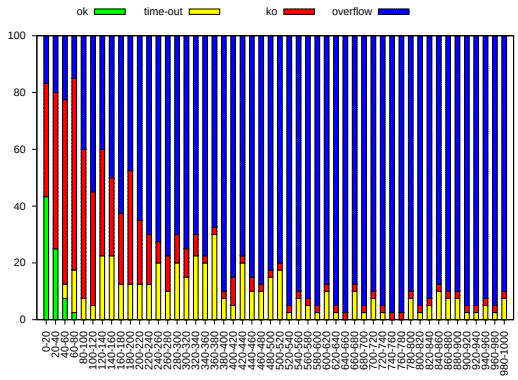
(a) FP



(b) gEDF



(c) gLLF



(d) LLREF

Figure 6.9 – Ratio d'acceptation en Uppaal avec variation du nombre de tâche et de précédences

tés aux graphiques 6.9. Les performances sont moins bonnes qu'avec les modèles C. Les modèles non ordonnançables sont plus difficiles à trouver, avec des temps de calcul plus élevés, comme on peut le voir sur les modèles gLLF et LLREF. De plus, pour toutes les politiques d'ordonnancement il y a une grande proportion d'overflow, c'est-à-dire dépassement des entiers d'Uppaal (limité aux entiers 16bits).

Nombre de processeurs Nous avons également regardé l'influence du nombre de processeurs sur le temps de calcul. La vérification est lancée pour des ensembles de 100 tâches asynchrones à échéances contraintes indépendantes, une charge processeur de 80%, une hyper-période de 10000, les périodes sont géométriques et il y a au plus 4 périodes différentes. On fait varier le nombre de processeur de 1 jusqu'à 100. Le test est lancé uniquement pour des modèles C.

La figure 6.10 montre que le temps augmente très raisonnablement (l'augmentation est d'approximativement 3 secondes entre 2 processeurs).

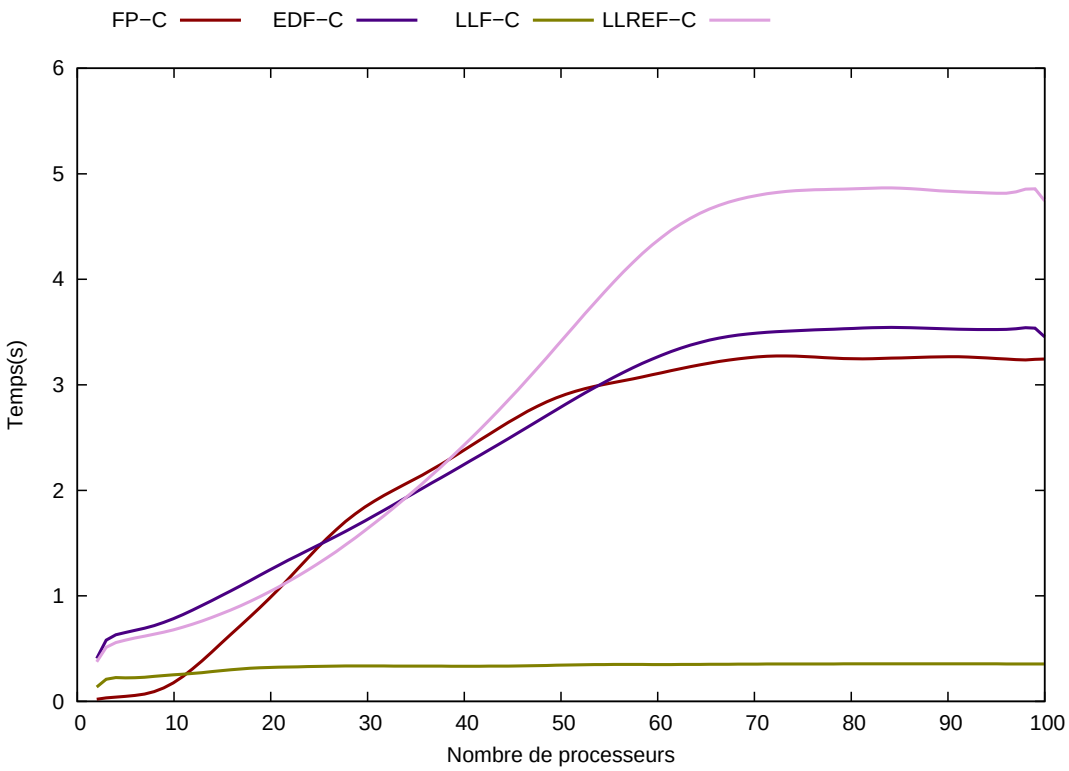


Figure 6.10 – Temps de calcul avec variation du nombre de processeurs

Valeur de l'hyper-période Nous avons également regardé l'influence de l'hyper-période. En effet, la profondeur de la recherche dépend directement de l'hyper-période donc plus celle-ci sera grande plus le temps de calcul sera long. Le test a été réalisé sur des ensembles composés de 25 tâches asynchrones à échéances contraintes indépendantes, une charge processeur de 50%, les périodes sont géométriques et il y a au plus 4 périodes différentes. On fait varier l'hyper-période avec un pas de 25 et pour chaque pas on génère 25 ensembles de tâches. Les résultats sont montrés dans la figure 6.11. politique donnée.

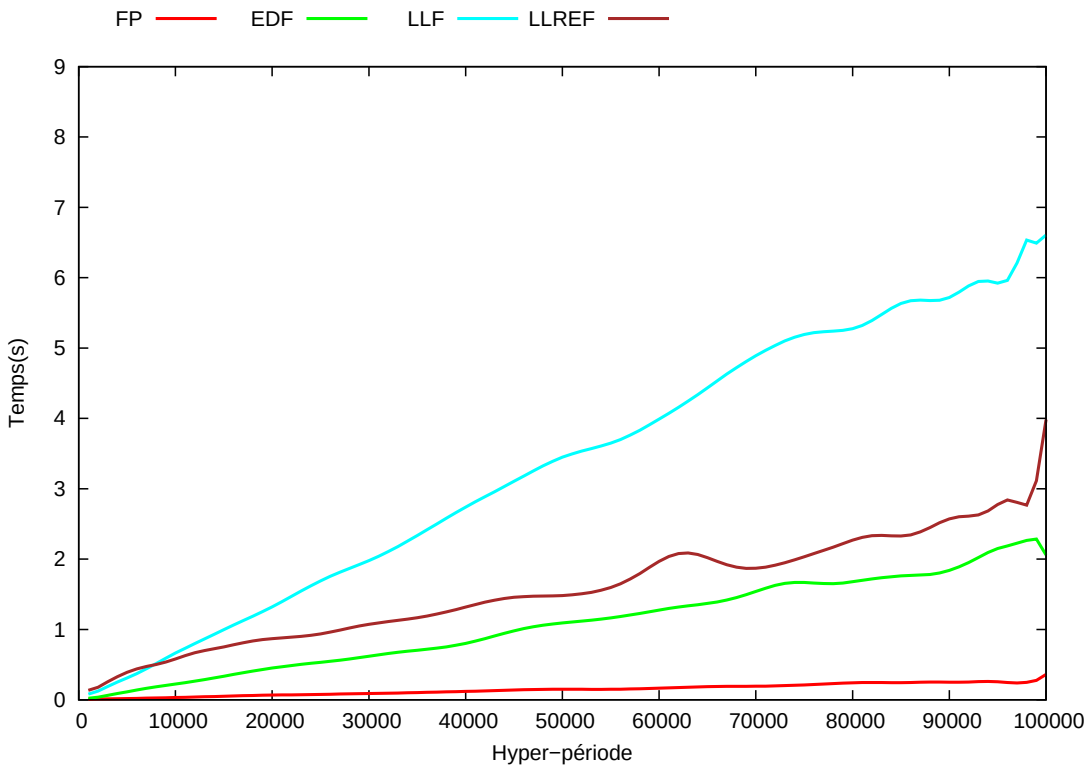


Figure 6.11 – Temps de calcul avec variation de l'hyper-période (en C uniquement)

Le fait d'avoir pris une charge à 50% assure qu'il y a un bon ratio d'acceptation pour toutes les politiques (voir les tests sur le paramètre de la charge page 107). Sur la figure, ne sont représentés que les temps de vérification des ensembles ordonnançables. Les temps d'analyse de FP sont très bons, quelques soit l'hyper-période. Les autres politiques ont des courbes linéaires.

Modèle optimisé Dans la section 15, nous avons proposé un modèle du système simplifié. Dans la graphique 6.12 nous montrons une comparaison entre le temps de calcul du modèle normal et celui du modèle optimisé. Le jeu de tâches utilisé pour l'analyse est celui utilisé dans le graphique 6.3. Le modèle optimisé réalise la recherche plus rapidement et on constate à partir de 300 tâches qu'il est quasiment deux fois plus rapide que le modèle normal excepté pour gEDF où les temps de calcul sont très similaires.

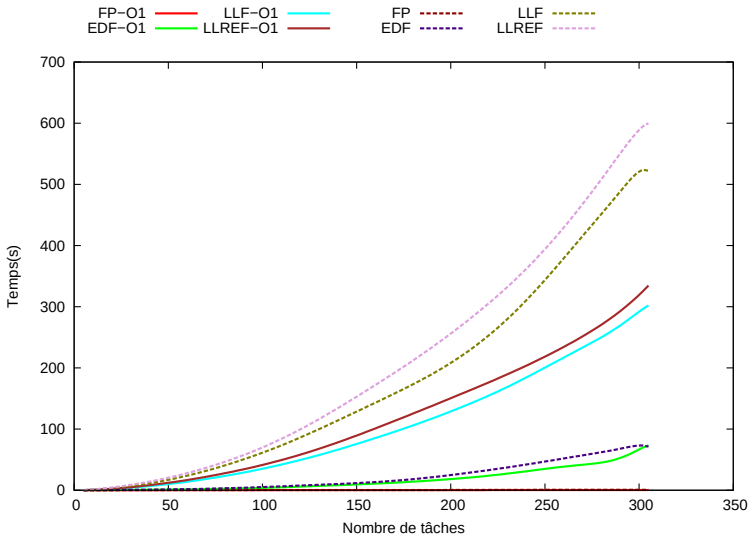


Figure 6.12 – Temps de calcul version optimisée avec variation du nombre de tâches

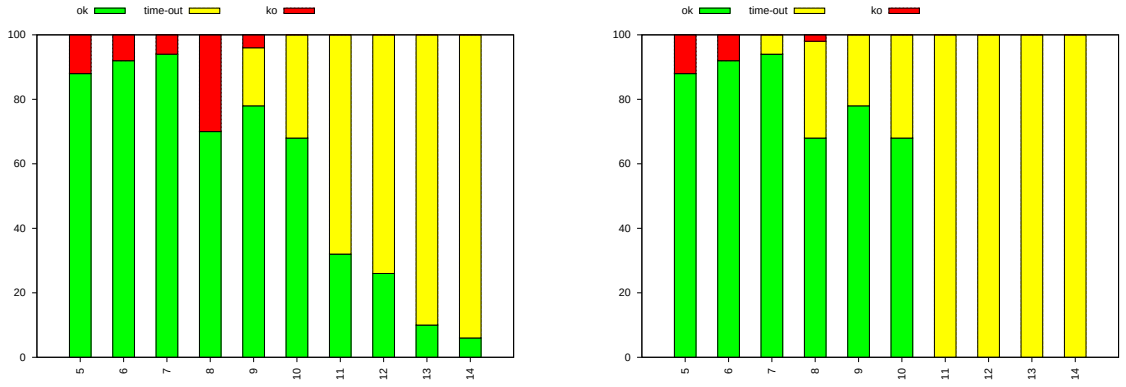
6.2.4 Expérimentation de la génération de paramètres hors-ligne

Nous avons également analysé les ratios d'acceptation et les performances temporelles pour la recherche des paramètres hors-ligne.

Performances de la recherche d'une affectation statique de priorités hors-ligne en C

La première série d'expérimentations a porté sur la recherche force brute d'une affectation valide de priorité pour une politique FP. Les résultats sont montrés dans les histogrammes de la figure 6.14. Pour cette série de tests, nous avons généré des ensembles de 5 et 15 tâches. On fait varier ce nombre par un pas de 1. Pour chaque pas, nous générons 50 ensembles de tâches avec un hyper-période de 1000.

Les résultats obtenus sont très limités. Pour la version C, le modèle arrive à trouver des ensembles ordonnançables jusqu'à 15 mais à partir de 11 le ratio est très bas (moins du 40%). La version Uppaal est plus limitée et à partir de 10 on n'arrive pas à décider sur aucun ensemble.



(a) Test en C

(b) Test en Uppaal

Figure 6.13 – Affectation statique de priorités hors-ligne

Performances de la recherche d'une affectation statique de priorités hors-ligne avec l'algorithme sous-optimal

Dans la page 72 nous présentons un algorithme sous-optimal pour l'affectation de priorités hors-ligne. Nous avons étudié son ratio d'acceptation avec le jeu de tâches utilisé pour analyser la taille des ensembles du graphique 6.3. Son taux de réussite est élevé et peut se comparer avec celui des politiques gEDF et LLREF. Toutefois, des time-out apparaissent à partir de 100-120 tâches et augmentent progressivement.

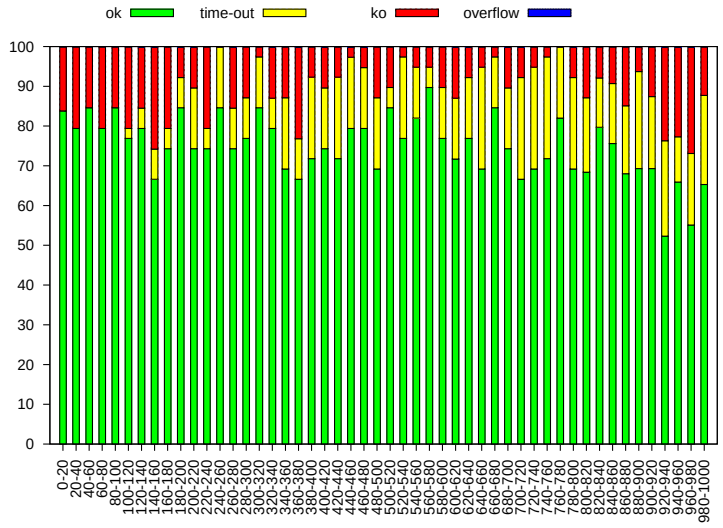


FIGURE 6.14 – Affectation statique de priorités hors-ligne par heuristique

Performances de la recherche d'un ordonnancement optimal hors-ligne en UPPAAL

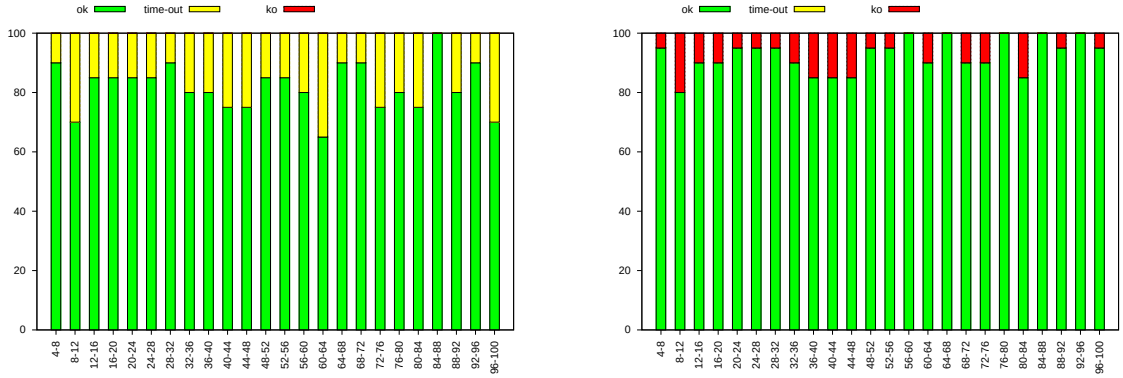
Pour mesurer les performances de la recherche d'un ordonnancement optimal, nous réalisons une série de recherches et nous montrons dans la graphique 6.15(a) le pourcentage d'ordonnancements trouvés. Pour cela, nous générons des ensembles à nombre de tâche variable entre 4 et 100 et un pas de 1. Pour chaque pas nous générons 25 tâches d'une hyper-période de 4000. Nous cherchons un ordonnancement optimal (le premier trouvé avec un algorithme de recherche en profondeur) pour une architecture à 2 processeurs.

Le taux de réussite se trouve, dans tous les cas, entre 60% et 100%. Dans la figure 6.15(b) on montre les ratios d'acceptation pour gLLF sur le même jeu de tâches. On peut noter que les résultats sont très proches sauf que gLLF conclut à chaque fois, notamment dans le cas des ensembles non ordonnançables.

Cependant, cette méthode est très limitée à cause de l'explosion combinatoire. Ainsi, en augmentant l'hyper-période ou le nombre de tâches, on tombe rapidement sur des ensembles non vérifiables (pour un jeu de tâches similaire avec une hyper-période de 20000 le calcul de séquence n'arrive à conclure sur aucun ensemble).

6.3 Étude de cas

Dans cette section, nous présentons la vérification et l'exécution du système de navigation simplifié FAS présenté dans l'introduction, page xi. Cet exemple illustre la chaîne complète de développement d'une application temps réel utilisateur.



(a) Optimal (b) LLF

Figure 6.15 – Calcul d'une séquence hors-ligne

6.3.1 Spécification en Prelude

La première phase pour l'utilisateur est de spécifier le système complet à l'aide du langage Prelude. Une solution est décrite ci-dessous.

```
imported node Gyro_Acq(gyro, tc: int) returns (o: int) wcet 10;
imported node GPS_Acq(gps, tc: int) returns (o: int) wcet 10;
imported node FDIR(gyr, gps, str, gnc: int) returns (to_pde,to_gnc,to_tm: int) wcet 20;
imported node PDE(fdir, gnc: int) returns (pde_order: int) wcet 10;
imported node GNC_US(fdir, gyr, gps, str: int) returns (o: int) wcet 210;
imported node GNC_DS(us: int) returns (pde,sgs,pws: int) wcet 300;
imported node PWS(gnc: int) returns (pws_order: int) wcet 20;
imported node SGS(gnc: int) returns (sgs_order: int) wcet 20;
imported node Str_Acq(str,tc: int) returns (o: int) wcet 200;
imported node TM_TC(from_gr,fdir: int) returns (cmd: int) wcet 1000;

sensor gyro wcet 1; sensor gps wcet 1;
sensor str wcet 1; sensor tc wcet 1;
actuator pde wcet 1; actuator sgs wcet 1; actuator gnc wcet 1;
actuator pws wcet 1; actuator tm wcet 1;

node FAS(gyro: rate (100, 0); gps: rate (1000, 0);
         str:  rate (10000, 0); tc: rate (10000, 0))
returns (pde, sgs;gnc: due 300; pws, tm)
var gyro_acq, gps_acq, str_acq, fdir_pde,
    fdir_gnc, fdir_tm, gnc_pde, gnc_sgs, gnc_pws;
let
  gyro_acq = Gyro_Acq(gyro, (0 fby tm)*^100);
  gps_acq = GPS_Acq(gps, (0 fby tm)*^10);
  str_acq = Str_Acq(str, 0 fby tm);
  (fdir_pde, fdir_gnc, fdir_tm) =
     FDIR(gyro_acq, gps_acq*^10, (0 fby str_acq)*^100, (0 fby gnc)*^10);
  gnc=GNC_US(fdir_gnc/^10, gyro_acq/^10, gps_acq, str_acq*^10);
  (gnc_pde, gnc_sgs, gnc_pws)=GNC_DS(gnc);
  pde = PDE(fdir_pde, (0 fby gnc_pde)*^10);
  sgs = SGS(gnc_sgs);
  pws=PWS(gnc_pws~>1/2);
  tm = TM_TC(tc, fdir_tm/^100);
tel
```

Au début, sont déclarés tous les nœuds importés avec leur signature et leur wcet. Les entrées / sorties sont ensuite déclarées avec les mots clés `sensor` pour les entrées et `actuator` pour les sorties. PRELUDE requiert également un wcet pour les entrées/sorties. Vient ensuite la déclaration du nœud `FAS` : ses entrées ont des rythmes différents, par exemple `gyro` arrive avec une période de 10 sans décalage de phase. Une seule sortie est contrainte avec une échéance explicite : `gnc` doit être produit au plus tard à 300. Plusieurs variables intermédiaires sont déclarées. Ensuite, entre les mots clés `let` et `tel`, toutes les équations sont spécifiées. Cet exemple se trouve dans la distribution de SCHEDMCORE, dans le dossier `schedmcore/tests/prelude/FAS`. En utilisant le compilateur PRELUDE, on peut calculer automatiquement les horloges des expressions avec la commande :

```
preludec -node FAS -print_clocks FAS.plu
```

on obtient dans notre cas

FAS : ((100,0) * (1000,0) * (10000,0) * (10000,0))
-> ((100,0) * (1000,0) * (1000,0) * (1000,1/2) * (10000,0))

Après la compilation, les tâches temps réel obtenues et leurs paramètres sont montrés dans le tableau suivant :

τ_i	O_i	T_i	D_i	C_i
Gyro_Acq	0	100	100	10
GPS_Acq	0	1000	1000	10
FDIR	0	100	100	20
PDE	0	100	100	10
GNC_US	0	1000	1000	210
GNC_DS	0	1000	1000	300
PWS	500	1000	1000	20
SGS	0	1000	1000	20
Str_Acq	0	10000	10000	200
TM_TC	0	10000	10000	1000
gyro	0	100	100	1
gps	0	1000	1000	1
str	0	10000	10000	1
tc	0	10000	10000	1
pde	0	100	100	1
sgs	0	1000	1000	1
gnc	0	1000	1000	1
pws	500	1000	1000	1
tm	0	10000	10000	1

$\mathcal{R}$

$gyro \xrightarrow{(0,0)} Gyro_Acq$
$GNC_DS \xrightarrow{(0,0)} PWS$
$GNC_US \xrightarrow{(0,0)} GNC_DS$
$TM_TC \xrightarrow{(0,0)} tm$
$GNC_DS \xrightarrow{(0,0)} SGS$
$gps \xrightarrow{(0,0)} GPS_Acq$
$FDIR \xrightarrow{(0,0)} PDE$
$GNC_US \xrightarrow{(0,0)} gnc$
$PDE \xrightarrow{(0,0)} pde$
$GPS_Acq \xrightarrow{(0,0)} GNC_US$
$Str_Acq \xrightarrow{(0,0)} GNC_US$
$Gyro_Acq \xrightarrow{(0,0)} GNC_US$
$FDIR \xrightarrow{(0,0)} GNC_US$
$tc_US \xrightarrow{(0,0)} TM_TC$
$FDIR_US \xrightarrow{(0,0)} TM_TC$
$SGS \xrightarrow{(0,0)} sgs$
$str \xrightarrow{(0,0)} Str_Acq$
$PWS \xrightarrow{(0,0)} pws$
$GPS_Acq \xrightarrow{(0,0)} FDIR$
$Gyro_Acq \xrightarrow{(0,0)} FDIR$

6.3.2 Choix d'une politique d'ordonnancement adaptée

Après avoir spécifié le système, l'utilisateur doit choisir une politique adaptée à son système. Nous avons testé l'ordonnançabilité du programme pour toutes les politiques codées dans SCHEDMCORE CONVERTER sur une architecture composée de deux cœurs et nous obtenons les résultats suivants :

politique	ordonnançable ?	temps de calcul C	temps de calcul Uppaal
FP	Non ordonnançable	0.002s	0.085s
gEDF	Ordonnançable	0.013s	1.064s
gLLF	Ordonnançable	0.039s	2.332s
LLREF	Non ordonnançable	0.002s	0.240s
FP optimal	Non ordonnançable	3.034s	42.110s
optimal	Ordonnançable	-	3m10.142s

LLREF échoue à ordonnancer l'ensemble de tâches car il faudrait modifier les échéances des tâches prédécesseur. En effet, la politique va faire passer des tâches avec des grands l_τ au détriment de tâche dont les successeurs sont urgents.

6.3.3 Exécution réelle du FAS

On peut alors exécuter le FAS sur une cible réelle. Nous avons lancé l'exécution pour gEDF et nous obtenons la trace suivante avec un niveau de verbosité à 4 :

```
0| gyro | gps |
1| Gyro_Acq | GPS_Acq |
11| FDIR102 | str0 |
12| FDIR102 | tc0 |
13| FDIR102 | Str_Acq91 |
30| FDIR102 | Str_Acq91 |
31| PDE119 | TM_TC129 |
41| pde0 | TM_TC129 |
42| Str_Acq91 | TM_TC129 |
100| gyro0 | TM_TC129 |
101| Gyro_Acq80 | TM_TC129 |
111| FDIR102 | TM_TC129 |
131| PDE119 | TM_TC129 |
141| pde0 | TM_TC129 |
142| Str_Acq91 | TM_TC129 |
200| gyro0 | TM_TC129 |
201| Gyro_Acq80 | TM_TC129 |
211| FDIR102 | TM_TC129 |
231| PDE119 | TM_TC129 |
241| pde0 | TM_TC129 |
242| Str_Acq91 | TM_TC129 |
300| gyro0 | TM_TC129 |
301| Gyro_Acq80 | TM_TC129 |
...
```

La trace obtenue pour la séquence hors-ligne en Uppaal est exprimée sur 10503 unités de temps. Une fois compressée par l'outil SchedMCore Tracer, le fichier la contenant a une taille de 177,1 Ko.

```
[Time 1] -> gps tc
[Time 2] -> gyro GPS_Acq
[Time 3] -> str GPS_Acq
[Time 4] -> GPS_Acq Str_Acq
[Time 22] -> Str_Acq FDIR
[Time 42] -> TM_TC Str_Acq
[Time 90] -> PDE Str_Acq
[Time 100] -> Str_Acq
[Time 101] -> TM_TC Str_Acq
[Time 159] -> gyro Str_Acq
[Time 160] -> Gyro_Acq Str_Acq
[Time 170] -> Str_Acq FDIR
```

```
[Time 190] -> PDE Str_Acq
[Time 200] -> Str_Acq
[Time 201] -> TM_TC Str_Acq
[Time 204] -> GNC_US TM_TC
[Time 259] -> gyro TM_TC
[Time 260] -> Gyro_Acq TM_TC
[Time 270] -> TM_TC FDIR
[Time 290] -> PDE TM_TC
[Time 300] -> TM_TC
[Time 301] -> GNC_US TM_TC
...
```

6.4 Résumé

Dans ce chapitre nous avons présenté l'environnement de développement PRELUDE-SCHEDMCORE. Une utilisation dans le cycle de développement sur une petite étude de cas a été illustrée dans la partie 6.3. Cet environnement est parfaitement compatible avec le compilateur PRELUDE, permettant ainsi à l'utilisateur d'aller de la spécification jusqu'à l'exécution en vérifiant l'ordonnançabilité. L'utilisateur peut également spécifier son système avec un fichier descriptif de tâches. L'environnement SCHEDMCORE contient deux outils : CONVERTER gère la partie analyse formelle du comportement multithreadé et RUNNER gère la partie exécutive.

Nous avons également évalué les performances de l'outil CONVERTER en réalisant de nombreux tests en utilisant de la génération automatique d'ensembles de tâches. Les conclusions sont que l'analyse d'ordonnançabilité de politiques en-ligne marche très bien, même pour de « grands » ensembles de tâches. L'algorithme sous-optimal de recherche d'affectation de priorité obtient de bons ratios d'acceptation. La méthode force brute et la recherche de séquence optimale sont de jolis résultats théoriques mais qui dans la pratique ne peuvent être utilisées, tout du moins en l'état.

Conclusions et perspectives

Résumé

Nous avons développé durant cette thèse une boîte à outils, SchedMCore, permettant la vérification formelle de l'ordonnançabilité et l'exécution multithreadée sur une cible multicœur de systèmes temps réel critiques. Les systèmes sont multipériodiques, à démarrage différé (asynchrone), à échéances contraintes, contraints par des précédences généralisées et communicant selon des motifs de communication précis. La description du système dans l'environnement SchedMCore peut se faire actuellement par un simple fichier descriptif des attributs temps réel ou par une spécification complète en Prelude.

L'analyse d'ordonnançabilité est une extension et une généralisation de plusieurs travaux existants sur le sujet. La vérification proposée est basée sur le parcours exhaustif du comportement avec pas de temps discret. Plusieurs analyses sont intégrées :

1. analyse d'ordonnançabilité par des politiques en-ligne, aujourd'hui sont proposées FP, gEDF, gLLF et LLREF. Les ordonnanceurs considérés sont des politiques globales, c'est-à-dire préemptives et à migration complète. Nous avons également étudié les comportements déterministes et non déterministes des politiques. De plus, de part la conception modulaire de l'outil, l'utilisateur peut facilement ajouter une politique supplémentaire ;
2. calcul d'une affectation de priorité valide en priorité fixe. Nous avons proposé une implantation de l'algorithme brute force et un algorithme sous-optimal qui obtient de bons ratios d'acceptation ;
3. calcul d'une séquence valide hors-ligne. La nouveauté sur le calcul de séquence concerne le traitement d'ensembles de tâches asynchrones.

L'outil associé à l'analyse d'ordonnançabilité est SchedMCore Converter. Les deux premiers algorithmes ont été implantés en C et avec le model checker Uppaal. Le dernier n'est implanté qu'en Uppaal. Un des problèmes concerne la terminaison du parcours : en effet, il n'existe aucun résultat théorique à l'heure actuelle sur la fenêtre d'analyse de systèmes asynchrones ou dépendants. L'utilisation d'un model checker permet de s'abstraire de cette connaissance, puisque l'outil est conçu pour reconnaître des états déjà rencontrés. En C, nous avons sauvé des configurations à hyper-période pas de temps ce qui fonctionne très bien dans la pratique. L'ensemble des expérimentations menées montre que l'approche est utilisable pour des ensembles de tâches réalistes.

La deuxième contribution de cette thèse concerne le dévelopement de la plateforme SchedMCore Runner. Il s'agit d'un exécutif qui permet l'exécution multithreadée des ensembles de tâches temps réel sur des architectures multicœur. Les politiques proposées dans Runner sont les mêmes que celles de l'analyse d'ordonnançabilité, à savoir pour les en-ligne FP, gEDF, gLLF et LLREF. L'outil prend en également en entrée des séquences valides générées dans l'algorithme 3 en Uppaal. L'exécutif permet 3 modes d'utilisation : allant de la simulation temporelle à

l'exécution temps précis des comportements des tâches. Il est compatible POSIX et facilement portable sur divers OS.

Perspectives

Tests de non régression Nous avons commencé à mettre en place des tests de non régression de façon à faciliter les modifications du code de SCHEDMCORE. Ce travail reste encore assez préliminaire et est primordial pour le maintien opérationnel d'un outil informatique.

Recherche de l'ordonnanceur optimal L'algorithme qui rencontre rapidement le problème de l'explosion combinatoire est celui concernant la recherche d'un ordonnanceur optimal. Dans un premier temps, nous allons étendre le modèle optimisé présenté en 15 sur la recherche de séquence optimale. Nous avons commencé à traduire les modèles dans d'autres model checker et celui qui a montré de bonnes performances est l'environnement FIACRE [BBF+08] / TINA [32] [BV06]. Nous avons commencé à travailler avec Bernard Berthomieu et Alexandre Hamez pour proposer dans SCHEDMCORE une branche traduisant les modèles en FIACRE (aussi bien pour le cas optimal que dans le cas des autres analyses d'ordonnançabilité).

Une deuxième réflexion pourrait également être menée pour chercher des heuristiques et des solutions sous-optimales efficaces. Néanmoins, la question de fond reste celle de « l'intérêt » de cette méthode. En effet, dans nos expérimentations, nous n'avons pas réussi à générer automatiquement des ensembles de tâches faisables mais non ordonnançables par les politiques en-ligne (FP, gEDF, gLLF et LLREF). Il faudrait dans un premier temps essayer de caractériser ces tâches.

Extension de SCHEDMCORE à des manycœurs Dans cette thèse on a choisi les architectures multicœur comme cible pour la vérification et l'exécution des ensembles de tâches temps réel. La suite de ces travaux sera matérialisée par l'arrivée d'un post-doctorant dans l'équipe qui travaillera sur l'adaptation de SCHEDMCORE pour des architectures manycœurs. Cette migration modifiera le paradigme de communication : en effet, les variables ne seront plus échangées par mémoire partagée mais par envoi de messages explicites. Cette extension aura un impact aussi bien sur l'analyse que sur l'exécutif. Il faudra notamment modifier le modèle de vérification de SCHEDMCORE CONVERTER pour prendre en compte entre autre les messages échangés et les temps de communication. Concernant, l'exécutif SCHEDMCORE RUNNER, il faudra utiliser les primitives système du manycœur.

Analyse détaillé des politiques Les résultats obtenus dans le chapitre 6.2 montrent que la politique gLLF offre d'excellents ratios d'acceptation. Toutefois, la vérification ne prend pas en compte les changements de contexte et les migrations. Dans une exécution réelle, ces deux effets peuvent produire de dépassement d'échéances. Une amélioration possible consisterait à analyser ces deux facteurs et offrir à l'utilisateur un rapport détaillé du comportement de façon à essayer de les réduire.

Sélection de traces d'exécution hors-ligne La trace obtenue à partir d'une recherche optimale correspond à la première trace ordonnançable trouvée avec un parcours en profondeur. Cependant, cette trace peut ne pas être la seule trace correcte pour l'ordonnancement de l'ensemble de tâches. Les caractéristiques de chaque trace peuvent être très différentes, comme par

32. http://homepages.laas.fr/bernard/tina/

exemple la taille, le nombre de changements de contexte et le nombre de migration. Selon les valeurs de ces trois paramètres l'exécution de la trace peut être faisable ou pas même si la trace est, en théorie, ordonnançable. C'est pour cela, qu'une possible amélioration de la recherche consisterait à optimiser la recherche par rapport à ces trois critères. Cette amélioration est limitée à des ensembles de tâches assez réduits étant donné la complexité d'exploration.

Adaptation à des systèmes non critiques La validation des systèmes critiques complique l'utilisation du SchedMCore Runner pour l'exécution. En revanche, d'autres systèmes plus flexibles au niveau temporel, comme les systèmes distribués ou temps réel souples, sont moins sensibles à la précision temporelle et accepterons un ordonnancement temps précis SchedMCore Runner.

Bibliographie

[ABJ01] Björn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *In Proc. 22nd IEEE Real-Time Systems Symposium*, pages 193–202. Society Press, 2001.

[AFM+02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. Times - a tool for modelling and implementation of embedded systems. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, pages 460–464, London, UK, UK, 2002. Springer-Verlag.

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

[Bak06] Theodore P. Baker. An analysis of fixed-priority schedulability on a multiprocessor. *Real-Time Syst.*, 32(1-2) :49–71, 2006.

[Bar95] Sanjoy K. Baruah. Fairness in periodic real-time scheduling. *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*, 0 :200–209, 1995.

[BB04] Enrico Bini and Giorgio C. Buttazzo. Biasing effects in schedulability measures. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 196–203, Washington, DC, USA, 2004. IEEE Computer Society.

[BB07] Theodore P. Baker and Sanjoy K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In *Handbook of Realtime and Embedded Systems*. CRC Press, 2007.

[BBF+08] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang, and François Vernadat. Fiacre : an Intermediate Language for Model Verification in the Topcased Environment. In *ERTS 2008*, Toulouse, France, 2008.

[BBFT06] J.-L. Bechennec, M. Briday, S. Faucou, and Y. Trinquet. Trampoline an open source implementation of the osek/vdx rtos specification. In *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, pages 62 –69, sept. 2006.

[BC07] Theodore P. Baker and Michele Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In *OPODIS*, pages 62–75, 2007.

[BC11] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5) :67–77, 2011.

[BCL09] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *IEEE Trans. Parallel Distrib. Syst.*, 20(4) :553–566, April 2009.

[BCPV94] Sanjoy K. Baruah, N. K. Cohen, C. Greg Plaxton, and Donald A. Varvel. Proportionate progress : A notion of fairness in resource allocation. *Algorithmica*, 15 :600–625, 1994.

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

[Bla77] Jacek Blazewicz. Scheduling dependent tasks with different arrival times to meet deadlines. In *Proceedings of the International Workshop organized by the Commision of the European Communities on Modelling and Performance Evaluation of Computer Systems*, pages 57–65, Amsterdam, The Netherlands, The Netherlands, 1977. North-Holland Publishing Co.

[BLGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations : the Signal language and its semantics. *Sci. of Compu. Prog.*, 16(2), 1991.

[BLR05] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32 :34–40, March 2005.

[BV06] Bernard Berthomieu and François Vernadat. Time petri nets analysis with tina. In *QEST*, pages 123–124. IEEE Computer Society, 2006.

[BWH93] A. Burns, A. Wellings, and A. Hutcheon. The impact of an ada run-time system's performance characteristics on scheduling models. In Michel Gauthier, editor, *Ada - Europe '93*, volume 688 of *Lecture Notes in Computer Science*, pages 240–248. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-56802-6_19.

[CBF+11] Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard, and Claire Pagetti. Developing critical embedded systems on multicore architectures : the Prelude-SchedMCore toolset. In *Proceedings of the 19th International Conference on Real-Time and Network Systems (RTNS'11)*, Nantes, France, September 2011. Irccyn.

[CBNP11] Mikel Cordovilla, Frédéric Boniol, Eric Noulard, and Claire Pagetti. Multiprocessor schedulability analyser. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC'11)*, 2011.

[CBNP12] Mikel Cordovilla, Frédéric Boniol, Eric Noulard, and Claire Pagetti. Off-line optimal multiprocessor scheduling of dependent periodic tasks. In *Proceedings of the 27th ACM Symposium on Applied Computing (SAC'12) - To appear*, 2012.

[CDKM02] Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri. *Scheduling in real-time systems.* John Wiley & Sons, October 2002.

[CFH+04] John Carpenter, Shelby Funk, Philip Holman, Anand Srinivasan, James Anderson, and Sanjoy K. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In *Handbook on Scheduling Algorithms, Methods, and Models.* Chapman Hall/CRC, Boca, 2004.

[CG06] Liliana Cucu and Joël Goossens. Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors. In *ETFA*, pages 397–404, 2006.

[CG07] Liliana Cucu and Joël Goossens. Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems. In *DATE '07 : Proceedings of the conference on Design, automation and test in Europe*, pages 1635–1640, San Jose, CA, USA, 2007. EDA Consortium.

[CGG11] Liliana Cucu-Grosjean and Joël Goossens. Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. *Journal of Systems Architecture*, 57(5) :561 – 569, 2011. Special Issue on Multiprocessor Real-time Scheduling.

[CLB+06] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. Litmusˆrt : A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*, pages 111–126, 2006.

[CRJ06] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *Proceedings of the 27th IEEE Real-Time Systems Symposium*, 0 :101–110, 2006.

[CSB90] Houssine Chetto, Marilyne Silly, and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2, 1990.

[Cur05] Adrian Curic. *Implementing Lustre programs on distributed platforms with real-time constraints*. PhD thesis, Université Joseph Fourier, Grenoble, 2005.

[CYC+05] Youngchul Cho, Sungjoo Yoo, Kiyoung Choi, Nacer-Eddine Zergainoh, and Ahmed Amine Jerraya. Scheduler implementation in mp soc design. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, ASP-DAC '05, pages 151–156, New York, NY, USA, 2005. ACM.

[DB09] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems. techreport YCS-2009-443, University of York, Department of Computer Science, 2009.

[DILS10] Alexandre David, Jacob Illum, Kim G. Larsen, and Arne Skou. *Model-Based Design for Embedded Systems*, chapter Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1, pages 93–119. CRC Press, 2010.

[DL78] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations research*, 1978.

[Dor08] Francois-Xavier Dormoy. Scade 6 a model based solution for safety critical software development. In *Embedded Real-Time Systems Conference (2008)*, 2008.

[FBG+10] Julien Forget, Frédéric Boniol, E. Grolleau, D. Lesens, and Claire Pagetti. Scheduling dependent periodic tasks without synchronization mechanisms. *In 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*, April 2010.

[FBLP08] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A multi-periodic synchronous data-flow language. In *11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*, Nanjing, China, December 2008.

[FBLP10] Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A real-time architecture design language for multi-rate embedded control systems. In *25th ACM Symposium On Applied Computing (SAC'10)*, Sierre, Switzerland, March 2010.

[FCTS09] Dario Faggioli, Fabio Checconi, Michael Trimarchi, and Claudio Scordino. An edf scheduling class for the linux kernel. In *Proceedings of the* 11th *Real-Time Linux Workshop (RTLWS 2009)*, Dresden, Germany, October 2009.

[Fer11] Jean-Michel Ferrard. Permutations d'un ensemble fini (programmation avec maple), préparation à la nouvelle épreuve d'informatique. Technical report, Ecole Polytechnique, 2011.

[FGPR11] Julien Forget, Emmanuel Grolleau, Claire Pagetti, and Pascal Richard. Dynamic priority scheduling of periodic tasks with extended precedences. In *IEEE International Conference on Emerging Technology and Factory Automation (ETFA'11)*, Toulouse, France, 2011.

[FKPY07] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata : Schedulability, decidability and undecidability. *International Journal of Information and Computation*, June 2007.

[For09] Julien Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. PhD thesis, Université de Toulouse - ISAE/ONERA, Toulouse, France, November 2009.

[FR92] Dror G. Feitelson and Larry Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *Journal of Parallel and Distributed Computing*, 16 :306–318, 1992.

[FY04] Elena Fersman and Wang Yi. A generic approach to schedulability analysis of real-time tasks. *Nordic J. of Computing*, 11(2) :129–147, 2004.

[Gal95] Bill O. Gallmeister. *POSIX.4 : programming for the real world*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1995.

[GCG00] Emmanuel Grolleau and Annie Choquet-Geniet. Off-line computation of real-time schedules by means of petri nets. In *Workshop On Discrete Event Systems, WODES2000*, Discrete Event Systems : Analysis and Control, pages 309–316, Ghent, Belgium, 2000. Kluwer Academic Publishers.

[GGD+07] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao, and Ge Yu. Exact schedulability analysis for static-priority global multiprocessor scheduling using model-checking. In *Proceedings of th Software Technologies for Embedded and Ubiquitous Systems, 5th IFIP WG 10.2 International Workshop (SEUS'07)*, pages 263–272, 2007.

[GGJY76] M. R. Garey, Ronald L. Graham, David S. Johnson, and A. C. Yao. Resource constrained scheduling as generalized bin packing. *Journal of Combinatorial Theory*, 21 :257–298, 1976.

[GGL+08] Nan Guan, Zonghua Gu, Mingsong Lv, Qingxu Deng, and Ge Yu. Schedulability analysis of global fixed-priority or edf multiprocessor scheduling with symbolic model-checking. In *ISORC*, pages 556–560, 2008.

[Gra69] Ronald Lewis Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17 :416–429, 1969.

[Gro99] Emmanuel Grolleau. *Ordonnancement temps réel hors-ligne optimal à l'aide de réseaux de Petri en environnement monoprocesseur et multiprocesseur*. PhD thesis, LISI-ENSMA, 1999.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[HL88] Kwang Soo Hong and Joseph Y.-T. Leung. On-line scheduling of real-time tasks. *IEEE Transactions on Computers*, 41 :1326–1331, 1988.

[HL93] Rhan Ha and Jane W.S. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1993.

[IEE] IEEE. IEEE POSIX Certification Authority. http ://standards.ieee.org/regauth/posix/.

[LD60] A. H. Land and A. G Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3) :497–520, 1960.

[LGDG00] José María López, Manuel García, José Luis Díaz, and Daniel F. García. Worst-case utilization bound for edf scheduling on real-time multiprocessor systems. In *ECRTS*, pages 25–33, 2000.

[LGG11] Markus Lindström, Gilles Geeraerts, and Joël Goossens. A faster exact multiprocessor schedulability test for sporadic tasks. *CoRR*, abs/1105.5055, 2011.

[LL73] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1) :46–61, 1973.

[LM80] Joseph Y.-T. Leung and M. L. Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Inf. Process. Lett.*, 11(3) :115–118, 1980.

[LMM98] Sylvain Lauzac, Rami Melhem, and Daniel Mosse. Comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor. In *In 10th Euromicro Workshop on Real Time Systems*, pages 188–195, 1998.

[LR09] Didier Lime and Olivier H. Roux. Formal verification of real-time systems with preemptive scheduling. *Real-Time Syst.*, 41(2) :118–151, 2009.

[LRSF04] Peng Li, Binoy Ravindran, Syed Suhaib, and Shahrooz Feizabadi. A formally verified application-level framework for real-time scheduling on posix real-time operating systems. *IEEE Trans. Softw. Eng.*, 30 :613–629, September 2004.

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

[MKK+10] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite : A distributed parallel simulator for multicores. In *HPCA*, pages 1–12, 2010.

[Mok83] Aloysius Ka-Lau Mok. Fundamental design problems of distributed systems for the hard real-time environment. Technical Report 297, Massachusetts Institute of Technology, Cambridge, MA, USA, may 1983.

[PFB+11] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3) :307–338, 2011.

[Pou06] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, 2006.

[RH01] Mario Aldea Rivas and Michael González Harbour. Marte os : An ada kernel for real-time embedded applications. In *In Proceedings of the International Conference on Reliable Software Technologies, Ada-Europe-2001*, 2001.

[RH02] Mario Aldea Rivas and Michael González Harbour. POSIX-Compatible Application-Defined Scheduling in MaRTE OS. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS'02)*, pages 67–75, Washington, USA, 2002.

[SAA+04] Lui Sha, Tarek Abdelzaher, Karl-Erik AArzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory : A historical perspective. *Real-Time Syst.*, 28 :101–155, November 2004.

[SG91] Terry Shepard and J. A. Martin Gagné. A pre-run-time scheduling algorithm for hard real-time systems. *IEEE Trans. Softw. Eng.*, 17 :669–677, July 1991.

[SSNB94] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE COMPUTER*, 28 :16–25, 1994.

[STC06] Christos Sofronis, Stavros Tripakis, and Paul Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Proceedings of the 6th International Conference on Embedded Software (EMSOFT'06)*, pages 21–33, Seoul, South Korea, October 2006.

[UCS+10] Theo Ungerer, Francisco J. Cazorla, Pascal Sainrat, Guillem Bernat, Zlatko Petrov, Hugues Cassé, Christine Rochange, Eduardo Quinones, Sascha Uhrig, Mike Gerdesa, Irakli Guliashvili, Michael Houston, Florian Kluge, Stefan Metzlaff, JÃ¶rg Mische, Marco Paolieri, and Julian Wolf. MERASA : Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro*, 30(5) :66–75, September/October 2010.

[UDT10] Richard Urunuela, Anne Marie Déplanche, and Yvon Trinquet. Storm a simulation tool for real-time multiprocessor scheduling evaluation. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1 –8, sept. 2010.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7 :36 :1–36 :53, May 2008.

[XP90] Jia Xu and David Parnas. Scheduling processes with release times, deadlines, precedence and exclusion relations. *IEEE Trans. Softw. Eng.*, 16 :360–369, March 1990.

[XP93] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transaction Software Engineering*, 19 :70–84, January 1993.