Programmation sûre de plates-formes embarquées de type multi/pluri-cœurs

Claire Pagetti

To cite this version:

Claire Pagetti. Programmation sûre de plates-formes embarquées de type multi/pluri-cœurs. Langage de programmation [cs.PL]. INPT, 2015. tel-01247922

HAL Id: tel-01247922
https://hal.science/tel-01247922

Submitted on 23 Dec 2015

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.

HDR

Thèse d'habilitation à diriger des recherches

Délivré par *l'INPT*

Spécialité *Informatique*

Présentée et soutenue par

Claire Pagetti

Le *29 janvier 2015*

Programmation sûre de plates-formes embarquées de type multi/pluri-cœurs

---

JURY

Rapporteur	Olivier H. Roux	Professeur, IRCCyN Nantes
Rapporteur	Frank Singhoff	Professeur, Université de Brest
Rapporteur	Jean-Pierre Talpin	Directeur de Recherches, INRIA Rennes
Examinatrice	Florence Maraninchi	Professeur des Universités, INPG / ENSIMAG
Examinatrice	Françoise Simonot-Lion	Professeur émérite, INPL - Ecole des Mines de Nancy
Examinateur	Yves Sorel	Directeur de recherche, INRIA Rocquencourt
Correspondant	Christian Fraboul	Professeur des universités, INPT-ENSEEIHT

École Doctorale MITT

Mathématiques Informatique et Télécommunication de Toulouse

Table des matières

1 Introduction **1**
1.1 Contexte . . . 1
1.1.1 Exemple d'application embarquée de type contrôle/commande . . . 1
1.1.2 Architectures modernes de processeurs . . . 2
1.1.3 Conception d'applications embarquées - limites de l'existant . . . 4
1.2 Contributions . . . 5
1.2.1 Etape 1 : définition d'un langage formel d'assemblage Prelude . . . 6
1.2.2 Etape 2 : recherche d'un placement prédictible sur la cible . . . 6
1.2.3 Etape 3 : développement d'exécutifs bare-metal prédictibles pour multi/ pluri-cœurs 7
1.2.4 Intégration dans un système à criticité mixte . . . 8
1.3 Organisation du document . . . 8

2 Prelude : langage de spécification de systèmes multi-périodiques **9**
2.1 Contexte . . . 9
2.1.1 Besoins et objectifs . . . 9
2.1.2 Travaux connexes . . . 10
2.2 Présentation du langage d'assemblage . . . 11
2.2.1 Description de la syntaxe Prelude . . . 11
2.2.2 Sémantique . . . 14
2.3 Génération de code multi-tâches . . . 16
2.3.1 Ensemble de tâches temps réel . . . 17
2.3.2 Protocole de communication . . . 19
2.3.3 Modèle de tâches générées . . . 21
2.3.4 Implantation mono-processeur sans sémaphore . . . 21
2.4 Utilisation de Prelude sur le cas d'étude Rosace . . . 22
2.5 Vers une spécification simplifiée . . . 26
2.5.1 Latence de chaînes fonctionnelles . . . 27
2.5.2 Assemblages mono-périodiques avec l'opérateur « don't care » . . . 28
2.5.3 Travaux connexes. . . . 32
2.6 Conclusion . . . 33

3 Modèles d'exécution prédictibles pour multi/pluri-cœurs **35**
3.1 Contexte . . . 35
3.1.1 Besoins et objectifs . . . 35
3.1.2 Travaux connexes . . . 37
3.2 SchedMCore : environnement de simulation/exécution de systèmes multi-périodiques . 38
3.2.1 Encodage de l'évolution des tâches . . . 39
3.2.2 Fenêtre de faisabilité . . . 41
3.3 Modèle d'exécution 1 : par tranches fines . . . 42
3.3.1 Validation du modèle . . . 42
3.3.2 Calcul du WCET d'une tranche d'exécution . . . 43
3.3.3 Calcul du WCTT d'une tranche de communication . . . 44
3.4 Modèle d'exécution 2 : AER . . . 48
3.5 Modèle d'exécution 3 : contentions réduites . . . 50
3.6 Conclusion . . . 51

4 Implantations prédictibles sur multi/pluri-cœurs **53**
4.1 Contexte . . . 53
4.1.1 Besoins et objectifs . . . 53
4.1.2 Travaux connexes . . . 54
4.2 Techniques de placement de tâches . . . 55
4.2.1 Rappels . . . 55
4.2.2 Données du problème . . . 55
4.2.3 Formalisation du problème . . . 57
4.2.4 Contraintes pour les modèles d'exécution 2 et 3 . . . 58
4.2.5 Evaluation . . . 59
4.2.6 Cas du modèle d'exécution 1 . . . 60
4.3 Implantation bare-metal sur multi/pluri-cœurs . . . 62
4.3.1 Cibles multi/pluri-cœurs considérées . . . 62
4.3.2 Gestion temporelle . . . 64
4.3.3 Gestion des communications . . . 65
4.3.4 Détails sur les librairies . . . 66
4.4 Conclusion . . . 66

5 Exécution prédictible de systèmes à criticité mixte **69**
5.1 Contexte . . . 69
5.1.1 Besoins et objectifs . . . 69
5.1.2 Travaux connexes . . . 70
5.2 Approche . . . 71
5.2.1 Idée générale . . . 71
5.2.2 Représentation d'une tâche critique . . . 73
5.3 Phase hors-ligne . . . 75
5.3.1 Instrumentation d'une tâche critique . . . 75
5.3.2 Analyseur hors-ligne . . . 76
5.4 Phase en-ligne . . . 79
5.4.1 Calcul dynamique de $\mathsf{RWCET}_{\mathsf{iso/cri}}(x)$ . . . 79
5.4.2 Implantation logicielle . . . 79
5.5 Conclusion . . . 82

6 Bilan et perspectives **83**
6.1 Conclusion . . . 83
6.1.1 Bilan sur la programmation sûre de plates-formes embarquées de type multi/pluri-cœurs . . . 83
6.1.2 Thématiques non décrites dans le document . . . 84
6.2 Projet de recherche . . . 85
6.2.1 Vers une méthodologie de développement de bout en bout . . . 85
6.2.2 Langage, compilation et génération de configurations . . . 86
6.2.3 Architectures et exécutifs temps réel tolérants aux fautes . . . 87

7 CV détaillé **89**
7.1 Curriculum vitae . . . 89
7.2 Activités d'administration et responsabilités collectives . . . 90
7.3 Encadrement . . . 91
7.4 Enseignement . . . 94
7.5 Productions scientifiques . . . 94
7.5.1 Liste des publications . . . 94
7.5.2 Liste des présentations invitées . . . 99
7.5.3 Liste des rapports contractuels . . . 99

Chapitre 1

Introduction

Depuis 2005, je suis ingénieur de recherche à l'ONERA - Office National d'Etudes et de Recherches Aérospatiales - où le contexte de travail est un peu différent de la plupart des organismes de recherche. En tant qu'EPIC, le fonctionnement de l'organisme nécessite de trouver des sources de financement « avec un taux de couverture minimal ». De ce fait, nos choix scientifiques sont guidés non seulement par le contenu technique mais aussi par le financement associé et la conséquence est que nos travaux sont parfois un peu épars. J'ai du m'investir à mon arrivée dans plusieurs projets dont les thématiques variées étaient totalement nouvelles pour moi. J'ai par exemple travaillé :

— avec Christel Seguin sur un projet Airbus appelé *Depnet* dont l'objectif était de définir formellement la notion de *fiabilité opérationnelle* ;
— avec Christel Seguin et Muriel Brunet sur un projet FP7 appelé *Flysafe* pour lequel j'ai étudié les accidents avion liés à la foudre ;
— avec Jack Foisseau sur un projet interne afin de définir un langage formel de description de *systèmes de systèmes*.

Ces projets m'ont permis d'acquérir des connaissances de base dans le domaine aéronautique qui est celui de l'ONERA et étaient un très bon départ pour la suite de ma carrière. Depuis 2007, j'ai la chance de contribuer à des projets moins éloignés les uns des autres dont la ligne conductrice est la *programmation sûre de plates-formes embarquées*. Cette problématique reste néanmoins très vaste et je n'ai bien évidemment travaillé que sur un sous-ensemble des activités liées à de tels développements. L'objectif de ce document est de décrire une synthèse des travaux que j'ai menés autour de ce thème et de proposer des perspectives de recherche pour les années à venir.

1.1 Contexte

La programmation sûre des systèmes embarqués critiques est un enjeu industriel crucial dont l'importance ne va que croître dans les années à venir du fait de leur omniprésence dans nos vies. Mon champ de compétence porte sur les applications avioniques de type « contrôle/commande » et temps réel.

1.1.1 Exemple d'application embarquée de type contrôle/commande

Un exemple de contrôle/commande en avionique est le *système des commandes de vol*. Le système des commandes de vol, dites *primaires*, est constitué de l'ensemble des éléments entre le manche et les gouvernes destinés à contrôler l'attitude, la trajectoire et la vitesse de l'avion en mode de pilotage manuel. Le système de commandes de vol primaires comprend donc :

— les organes de pilotage : manche, palonnier, commandes de trim…
— les organes de transmission et de traitement des ordres de l'équipage, à savoir :
  - les timoneries et les câbles dans le cas de commandes mécaniques,
  - des calculateurs et des câblages dans le cas de commandes électriques,
— les actionneurs ou servocommandes permettant de positionner les gouvernes.

Au système des commandes de vol primaires est adjoint un système des commandes de vol secondaires gérant les gouvernes dites hypersustentatrices (les becs et les volets) utilisées à basses vitesses (atterrissages et décollages) pour augmenter la surface de la voilure et par conséquent sa portance.

Nous nous intéressons à l'implantation sûre du code associé aux commandes de vol électriques sur les calculateurs. Par conséquent, nous nous focalisons sur deux aspects : (1) la description de l'« application », c'est-à-dire la fonction exécutée par le calculateur (il s'agit donc d'un sous-ensemble du système) ; (2) la maîtrise du comportement réel sur le calculateur. Afin d'illustrer notre propos, considérons un exemple très simple de commandes de vol électriques primaires mais représentatif de la structure générale des applications avioniques visées comme nous le verrons avec les autres études de cas du document.

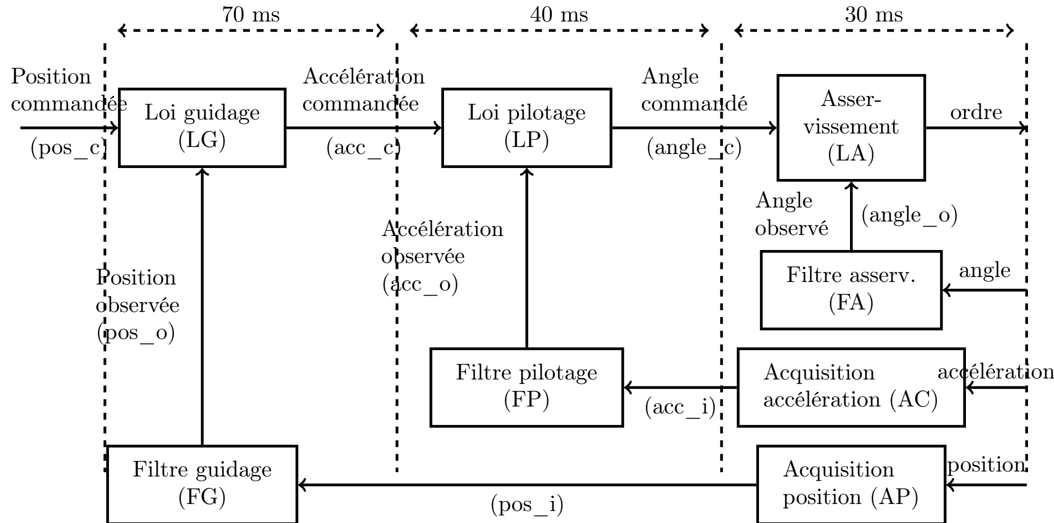


FIGURE 1.1 – Exemple d'application avionique de type contrôle/commande

La figure 1.1 montre l'architecture fonctionnelle de l'application : une boucle rapide à 30ms réalise les asservissements des gouvernes, la boucle intermédiaire à 40ms assure le pilotage automatique et la boucle la moins rapide gère le guidage de l'avion. Le guidage prend en entrée les consignes du pilote (*pos_ c*), les acquisitions récupèrent les données capteur (*acc, pos, angle*) et l'asservissement produit l'ordre (*ordre*) à envoyer aux actionneurs. Les tâches communiquent entre elles (quelle que soit leur période) et l'ordre des traitements est important pour la correction des calculs, de même que la fraîcheur des données consommées pour superviser les actionneurs. Pour un système réel de commandes de vol, il peut y avoir plus de 5000 fonctions à assembler (et non 8 comme dans l'exemple simplifié) qui ont été codées par des personnes différentes. L'*intégrateur système* a alors en charge d'assembler ces fonctions en respectant des contraintes globales et il doit également valider/vérifier le système réellement implanté.

Nous pouvons en conclure plusieurs choses sur les applications visées :

1. elles sont multi-périodiques, de grandes tailles et soumises à des contraintes temporelles fortes et de précédence. Il y a une activité déterminante consistant à assembler les fonctions ;
2. elles sont critiques, c'est-à-dire que leurs défaillances peuvent avoir des conséquences catastrophiques. En particulier, les délais introduits par le système informatique entre un stimulus extérieur et la réaction sur les gouvernes doivent être maîtrisés et bornés, un non respect de ces exigences temporelles pouvant provoquer la perte du contrôle.

1.1.2 Architectures modernes de processeurs

Une fois l'application développée, le code doit être exécuté sur une *architecture (ou plate-forme) avionique*. Il existe plusieurs types d'architectures, entre autre : (1) les *architectures fédérées* où chaque calculateur n'héberge qu'une unique application ; (2) les plates-formes *avioniques modulaires intégrées* (Integrated Modular Avionic - IMA) où les ressources sont partagées entre plusieurs applications. L'embarquement d'un composant, et en particulier d'un calculateur, n'est possible que s'il satisfait plusieurs propriétés selon plusieurs points de vue tels que sûreté de fonctionnement, temps réel, encombrement ou compatibilité électromagnétique. Mes travaux de recherche ont porté essentiellement sur la question du temps réel et ponctuellement sur des questions de sûreté de fonctionnement. Dans le travail présenté dans ce document, on suppose que l'architecture cible est un composant unique de type mono-processeur ou multi/pluri-cœurs, qui peut être partagé ou non par différentes applications avioniques.

Depuis le début des années 2000, on trouve sur le marché des PC grand public des processeurs multi-cœurs (quelques cœurs intégrés sur une même puce reliés par un bus partagé) et depuis 2010, des pluri-cœurs (nombreux cœurs intégrés sur une même carte connectés par un réseau sur puce) offrant des capacités de calcul impressionnantes et des rapports puissance de calcul / consommation électrique très bons. Sachant que les systèmes critiques n'embarquent que des technologies mâtures, aucun avion, satellite, lanceur ou système de régulation automobile n'utilisait ces technologies au début de mon travail. Aujourd'hui encore, leur utilisation est balbutiante et on trouve un premier pas dans l'automobile [CLMW14]. Les raisons de cet atermoiement sont dues à plusieurs facteurs [Gat13 ; Lap13] tels qu'une trop forte mortalité infantile, une mauvaise résistance aux vibrations ou des problèmes de validation temps réel. Les explications liées à ce dernier point (celui qui nous intéresse) en sont multiples :
Pour les multi-cœurs :

1. difficulté (voire impossibilité) de calculer les pires temps d'exécution (WCET, worst case execution time) d'une application. Les multi-cœurs sont majoritairement utilisés à des fins grand public où le critère de qualité est l'efficacité moyenne. L'enjeu dans le monde embarqué critique est de montrer la *prédictibilité*, c'est-à-dire d'assurer que dans tous les scénarios (et donc dans le pire cas), le temps nécessaire à l'exécution d'un programme est borné par un majorant. Or le matériel est optimisé pour la rapidité moyenne et contient des mécanismes très complexes (pipeline, prédiction de branchement, protocole d'accès au bus partagé) souvent peu documentés, ce qui complique voire rend impossible le calcul d'un majorant raisonnable.
2. difficulté de programmation. Tous les programmes et logiciels développés jusqu'à l'apparition des multi-cœurs tournaient sur un processeur unique et de ce fait suivaient le paradigme de programmation « séquentielle ». Les systèmes d'exploitation plus récents ont introduit la gestion multi-tâches en simulant un parallélisme entre les applications bien que la ressource processeur soit unique. Mais cet aspect multi-tâches ne remettait pas (ou peu) en cause la programmation séquentielle. L'arrivée de matériel vraiment parallèle nécessite soit de revoir les méthodes de programmation (en utilisant des langages tels que MPI, Open MP), soit de paralléliser (plus ou moins) automatiquement le code. Il n'existe à ce jour aucune réponse générique et facile à cette problématique.
3. (quasi) absence de systèmes d'exploitation temps réel critiques.

Pour les pluri-cœurs :

1. difficulté accrue de programmation. En effet, en plus des problèmes mentionnés auparavant, plusieurs mécanismes offerts matériellement par les multi-cœurs sont à la charge de l'utilisateur. En voici quelques exemples :
   — absence de cohérence de caches. Ce mécanisme sert à maintenir une vue globale unifiée pour tous les processeurs : si une donnée est dans les caches locaux de deux cœurs distants et qu'un des deux cœurs la modifie, le matériel invalidera l'autre cache. Ainsi lorsque le deuxième cœur souhaite accéder à la donnée, il la lit directement dans la mémoire RAM qui est également automatiquement mise à jour.
   — passage de message. Dans un multi-cœurs standard, les variables sont échangées par mémoire partagée. Dans un système avec un réseau sur puce, il est possible (1) soit de passer par la mémoire partagée (qui deviendra une zone de contention forte) ; (2) soit d'envoyer des messages entre cœurs. La deuxième solution nécessite davantage d'effort du programmeur.
   — routage des messages sur le réseau. L'utilisateur doit parfois spécifier les chemins pris par les paquets échangés entre cœurs, ou entre cœur et mémoire partagée.
2. absence de systèmes d'exploitation temps réel critiques.
3. manque de documentation détaillée.

Comparativement aux multi-cœurs, la prédictibilité des pluri-cœurs est plus aisée à appréhender et ce au détriment de la facilité de programmation. En effet, le fait de réduire les mécanismes matériels implicites est un atout du point de vue temps réel et prédictibilité. A titre d'exemple, la cohérence de cache est appréciable pour le programmeur qui ne doit pas gérer explicitement la mémoire mais est très néfaste d'un point de vue temps réel car elle génère des comportements matériels ainsi que des trafics sur le réseau interne implicites, donc non contrôlés temporellement. A l'inverse, lorsque les envois de messages sont explicites et réalisés par l'utilisateur, les moments d'accès au réseau, à certains périphériques et à la mémoire sont connus et l'un des avantages est d'éviter de saturer ces ressources partagées. Le choix du routage offre également la possibilité de réduire le nombre de contentions. En résumé, plus les puces ressemblent à des architectures distribuées, plus leur fonctionnement tend à être prédictible.

1.1.3 Conception d'applications embarquées - limites de l'existant

L'objectif naturel qui ressort des besoins industriels est de programmer de manière sûre des applications de type contrôle/commande sur des architectures modernes de type multi/pluri-cœurs. La conception de tels systèmes requiert en particulier trois activités :

1. spécifier formellement l'application et générer le code associé le plus automatiquement possible ;
2. offrir des outils de vérification formelle de propriétés au niveau spécification et implantation réelle, en particulier des outils de calcul de pire temps d'exécution WCET et d'analyse d'ordonnançabilité ;
3. fournir des exécutifs bas niveau *prédictibles* pour le calculateur cible.

Spécification

Plusieurs langages formels de haut niveau ont été définis afin de permettre la spécification de systèmes embarqués critiques, leur analyse et/ou leur génération de code automatique. Les langages synchrones [BCE03 ; And05], en particulier, sont depuis plusieurs années utilisés avec succès pour concevoir des applications de type contrôle/commande. L'environnement logiciel utilisé, entre autre par l'industrie aéronautique, est la suite Scade [Est12] qui permet de décrire graphiquement les programmes synchrones, vérifier des propriétés avec l'outil Prover et générer du code certifié selon le standard aéronautique DO 178 [RTC08 ; RTC11a].

Les langages synchrones sont parfaitement adaptés pour spécifier les sous-fonctions de la figure 1.1. En revanche, même s'ils permettent de décrire des assemblages multi-périodiques à l'aide d'horloges booléennes, cette méthode n'est pas vraiment appropriée pour cela comme nous l'avons montré dans [FBL08].

Une deuxième limite concerne le format de l'exécutable généré. En effet, les compilateurs actuels génèrent des codes séquentiels, ce qui donne lieu à des implantations mono-tâches. Hors depuis (1) l'adoption de systèmes d'exploitation embarqués critiques notamment OSEK [OSE05] pour l'automobile ou l'Arinc 653 [Aer97] pour l'aéronautique ; (2) l'émergence des nouvelles architectures multi/pluri-cœurs ; il est indispensable de proposer des codes multi-tâches parallélisables afin d'offrir une plus grande efficacité et une plus grande flexibilité dans l'intégration du code embarqué. Cependant, accepter des exécutions multi-tâches s'accompagne d'une nouvelle difficulté pour préserver la sémantique initiale : la liberté dans l'ordre d'exécution des tâches doit être maîtrisée de façon à respecter le flot de données initial, on parle alors de *déterminisme fonctionnel*.

Pour répondre à ces besoins, les avionneurs ont développé une sur-couche au dessus des langages synchrones permettant de décrire un système complet comme l'assemblage de sous-systèmes synchrones. Des formalismes « *maison* » ont vu le jour, tel que le langage ad hoc TEMPO utilisé par Airbus. C'est dans ce contexte et face à l'absence d'une sur-couche formelle décrivant l'assemblage temps réel de programmes de contrôle/commande que nous nous sommes penchés sur la définition du langage Prelude.

Vérification et validation

Le deuxième axe concerne la validation et la vérification de l'application développée aussi bien au niveau spécification qu'au niveau implantation réelle.

Le premier besoin est de vérifier les contraintes temporelles [RTC08]. Cela nécessite en particulier la capacité de prouver la prédictibilité de la cible et de calculer des pires temps d'exécution. Les avionneurs ont l'habitude d'embarquer des plates-formes de type COTS (*commercial off-the shelf component*, composant sur étagère), car aux yeux des autorités de certification, de telles architectures procurent une certaine confiance du fait de leur utilisation massive. Hors, il n'existe pas à ce jour de méthode générique permettant de calculer ces délais pires cas sur ces cibles contemporaines [WR12].

Le deuxième type d'analyse concerne l'*ordonnançabilité*, c'est-à-dire le respect des contraintes temporelles par l'ensemble des tâches exécutées. Un calcul de pire temps d'exécution détermine le nombre de cycles nécessaires à exécuter une tâche, tandis qu'une analyse d'ordonnançabilité combine l'ensemble des tâches et détermine les comportements entremêlés. Au début de nos travaux, peu de résultats étaient disponibles pour les applications multi-périodiques visées.

Une autre des exigences minimales imposées par la certification est de prouver le déterminisme du système [SAE10 ; RTC08], à savoir que le résultat est identique lorsqu'on applique plusieurs fois un même scénario. Il faut dès lors pouvoir valider le comportement fonctionnel à plusieurs niveaux du processus de développement, alors que dans la réalité, la plupart des analyses se font sur le système implanté avec des tests intensifs.

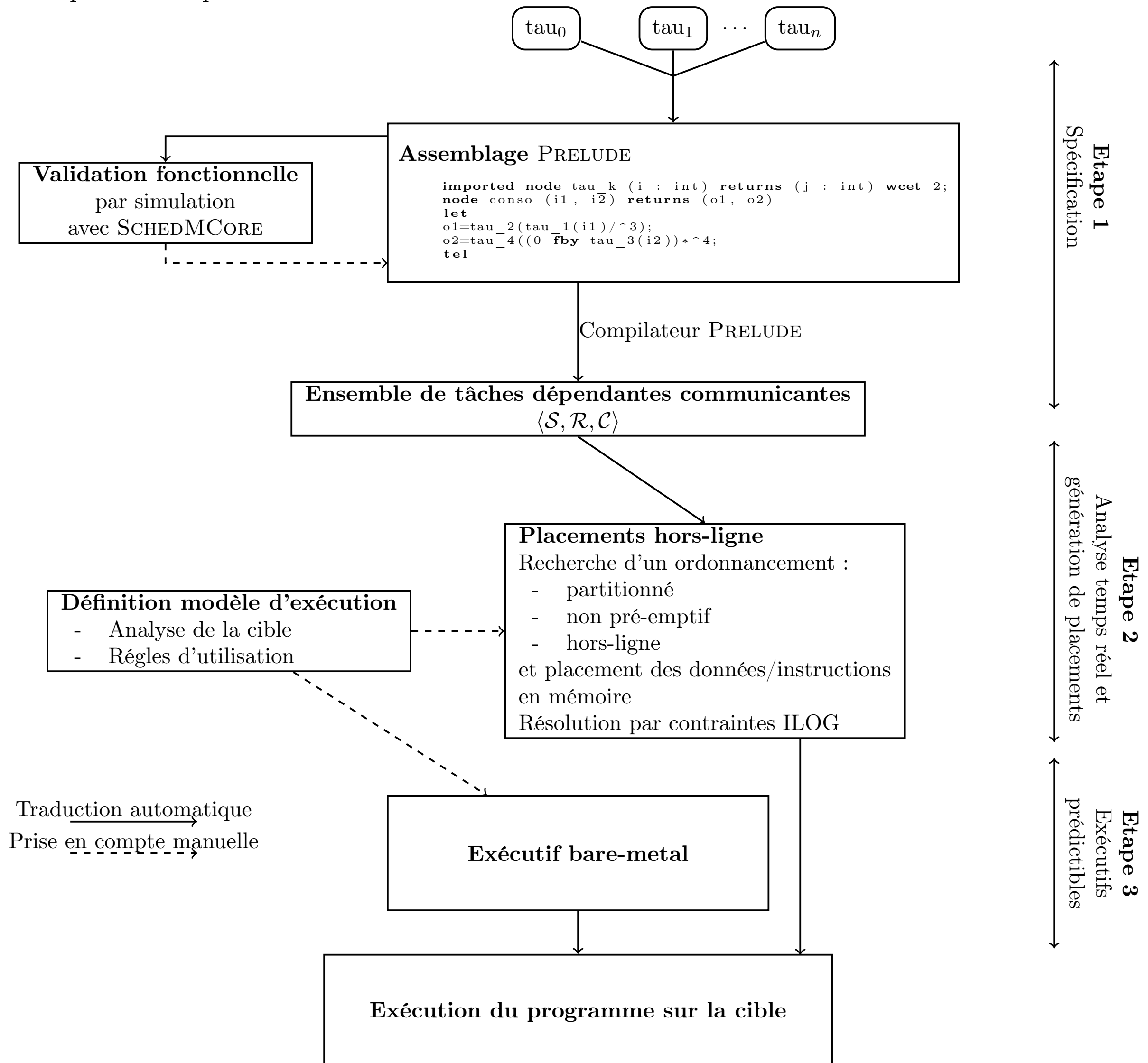
Ces différents constats nous ont amenés à réfléchir à des nouvelles approches méthodologiques en vue de maîtriser les processeurs COTS et de faciliter la phase d'intégration.

Exécution prédictible

Enfin, la dernière activité concerne l'implantation proprement dite sur le processeur COTS de façon à respecter les règles de prédictibilité et conserver les comportements fonctionnels des applications de type contrôle/commande. Il n'existe pas d'exécutifs ou d'hyperviseurs génériques répondant aux besoins de temps réel et de tolérance aux fautes pour les cibles envisagées. Cela tient au fait que les multi/pluri-cœurs maintiennent trop d'interdépendance entre les exécutions parallèles sur les différents cœurs et que les architectures matérielles proposées dans le commerce sont très (trop) variées. Ainsi, il est très difficile d'imaginer des solutions génériques à la fois dans la définition des règles d'utilisation mais également dans le codage. Dès lors, chaque exécutif doit être re-déployé pour chaque processeur et chaque nouvelle génération, entraînant des coûts non négligeables.

1.2 Contributions

Mes travaux d'encadrement et de recherche ont porté sur la programmation sûre d'assemblages multi-périodiques sur multi/pluri-cœurs. L'objectif est de fournir un environnement de développement continu, allant de la spécification jusqu'à l'implantation sur la cible. Cela nécessite plusieurs étapes dont l'organisation est illustrée dans la figure ci-dessous. Les flèches pleines représentent des transformations automatiques tandis que les hachurées sont des actions manuelles.



1.2.1 Etape 1 : définition d'un langage formel d'assemblage Prelude

Ce travail a été réalisé durant la thèse de Julien Forget (co-encadrée avec Frédéric Boniol) financée par EADS Astrium (David Lesens était notre correspondant technique). Le langage Prelude [FBLP08 ; For09] permet de décrire une application comme un assemblage multi-périodique de fonctions importées, supposées écrites dans un autre langage (Scade, C ou Ada). La base sémantique est celle du synchrone mais les horloges sont rythmées par des entiers représentant le temps réel et non plus des booléens. De plus, les flots sont calculés selon l'hypothèse *synchrone relâchée* [Cur05], c'est-à-dire avant la prochaine échéance.

Validation L'utilisateur peut valider le comportement fonctionnel de ses programmes par simulation en utilisant l'environnement SchedMCore. Cet environnement a été développé durant la thèse de Mikel Cordovilla (co-encadrée avec Frédéric Boniol et Eric Noulard) en support du compilateur Prelude. Un des objectifs est d'offrir un simulateur de programmes Prelude, dans l'idée du simulateur Luciole associé à Lustre.

Génération de code Une fois la spécification réalisée, le compilateur Prelude génère un ensemble de tâches périodiques dépendantes et communicantes. Le modèle de tâches considéré est très standard, puisque conforme à celui de Liu et Layland [LL73]. La nouveauté est l'introduction de précédences étendues et la gestion explicite des communications entre tâches. Le compilateur Prelude est disponible à l'url `http://www.lifl.fr/~forget/prelude.html`.

Extension de Prelude La thèse de Rémy Wyss (co-encadrée avec Frédéric Boniol) démarrée en 2010 consiste à étendre le langage Prelude pour lui ajouter davantage de flexibilité. L'idée est d'introduire un opérateur « don't care » pour déclarer des flots sans contraintes de précédence explicites. Le compilateur transforme alors cette spécification partielle en un programme Prelude déterministe respectant des contraintes de latence de bout en bout.

1.2.2 Etape 2 : recherche d'un placement prédictible sur la cible

A partir de l'ensemble de tâches généré par le compilateur Prelude, la deuxième activité consiste à intégrer de manière prédictible et la plus automatique possible les tâches sur la cible.

Ordonnancement avec une politique en-ligne Notre première approche a consisté à générer du code pour des systèmes d'exploitation temps réel :

- Marte OS [RH02] pour les cibles mono-processeurs, en évitant l'utilisation de sémaphores par codage des précédences étendues dans les dates de réveil et les échéances,
- notre exécutif maison SchedMCore, développé par Mikel Cordovilla, pour cibles multi-cœurs supportant les politiques globales pré-emptives (FP, gEDF, gLLF et LLREF). Cet exécutif nécessite du noyau sous-jacent quelques primitives de base et est également compatible Posix. SchedMCore permet également d'analyser l'ordonnançabilité des politiques exécutées avec l'hypothèse de connaître le WCET des tâches et que le coût des pré-emptions et migrations est nul.
- toujours dans la thèse de Mikel Cordovilla, nous avons tendu vers davantage de prédictibilité en générant des séquenceurs hors-ligne pré-emptifs globaux.

L'environnement SchedMCore est disponible à l'url `http://sites.onera.fr/schedmcore/`.

Définition de modèles d'exécution Les hypothèses utilisées dans la thèse de Mikel Cordovilla étant insuffisamment fortes pour assurer la prédictibilité, nous avons de ce fait orienté nos choix d'ordonnancement vers des séquenceurs partitionnés non pré-emptifs calculés hors-ligne. Nous avons de plus défini des règles supplémentaires assurant la calculabilité du WCET des tâches. En effet, pour forcer la prédictibilité sur des multi/pluri-cœurs, une solution est d'utiliser un *modèle d'exécution* adéquat [ABD13], c'est-à-dire un ensemble de règles à suivre par le concepteur lors de l'implantation de façon à éviter ou au moins réduire les comportements non prédictibles.

Nous avons établi trois modèles d'exécution pour multi-cœurs : le premier à base de tranches fines consiste à alterner des phases (1) d'*exécution* pendant lesquelles le cœur exécute localement son code en n'accédant qu'à ses caches locaux et jamais à la mémoire globale ; (2) de *communication* pendant lesquelles le cœur ne fait que des écritures et des lectures dans la mémoire globale. Le deuxième modèle

repose sur un stockage des données / instructions dans les caches locaux configurés en SRAM et sur des communications exécutées en isolation. Enfin, nous avons défini un modèle d'exécution pour pluri-cœurs : la réduction des contentions se fait par le stockage des données / instructions dans les mémoires locales et l'échange de données par passage explicite de messages.

Recherche automatique de placements Une fois les règles d'exécution définies, il faut placer l'application sur la cible, c'est-à-dire allouer les tâches sur les cœurs, placer les données / instructions dans la mémoire et calculer des séquencements non pré-emptifs hors-ligne, en tenant compte des instants de communication entre tâches. Nous sommes repartis de travaux de Gérard Bel (maître de recherche ONERA aujourd'hui à la retraite) dans le projet Airbus SCADE 2000. Il avait développé un prototype de calcul de séquence hors-ligne mono-processeur d'applications similaires à celles issues de PRELUDE reposant sur l'utilisation d'un solveur de contraintes. Nous avons poursuivi ses idées pour prendre en compte la mémoire, la communication de données entre tâches et la parallélisation.

1.2.3 Etape 3 : développement d'exécutifs bare-metal prédictibles pour multi/ pluri-cœurs

Pour implanter les modèles d'exécution précédents et face à l'absence d'exécutifs temps réel pour les multi/pluri-cœurs COTS considérés, nous avons développé des exécutifs « *bare-metal* » sur plusieurs cibles réelles. La programmation bare-metal consiste à programmer avec des primitives bas niveau afin d'éviter l'utilisation d'un système d'exploitation et ainsi obtenir des performances optimales, des comportements temps réel complètement maîtrisés, le tout avec des empreintes mémoires minimales.

Cibles multi/pluri-cœurs considérées Mon collègue Eric Noulard a initié nos recherches dans l'étude de l'embarquement de cibles multi/pluri-cœurs et a choisi la plupart de nos cibles. Sa première initiative en 2010 fut de nous inscrire dans la communauté MARC [1] (Many-core Applications Research Community) d'Intel et de nous donner accès au pluri-cœurs Single-chip Cloud Computer (SCC) [Int10 ; Int12]. La puce est composée de 48 cœurs connectés par un réseau sur puce. La description détaillée du processeur est donnée partie 4.3.1. Wolfgang Puffitsch, post-doctorant du projet RTRA TOAST, a porté l'environnement SCHEDMCORE et adapté le compilateur PRELUDE en conséquence.

Nous avons également investi dans l'achat d'un TILERA TILEmpower-Gx36 [Til13a]. Cette puce est composée de 36 cœurs, également connectés par un réseau sur puce, lequel est composé de cinq sous réseaux. Il existe plusieurs environnements de programmation, notamment Zero Overhead LINUX (ZOL) [Til13b, Chapter 7] qui isole les cœurs et les ressources locales. Nous avons mené des expérimentations sur la prédictibilité de cet environnement avec notre étudiant de master Romain Gratia et une partie de celles-ci est publiée dans [PSG14]. Les résultats, bien que bons dans l'ensemble, restent insuffisants pour nos besoins en prédictibilité. Des travaux sont en cours pour étendre nos librairies en mode *bare-metal*.

Plus récemment, nous nous sommes également équipés d'une plate-forme KALRAY [Kal12] composée de 256 cœurs répartis en 16 groupes de 16 cœurs. L'architecture d'un groupe est similaire à un multi-cœurs et les 16 groupes sont connectés par un réseau sur puce. Le développement d'un exécutif *bare-metal* pour cette cible est en cours dans la thèse de Quentin Perret (co-encadrée avec Eric Noulard et Pascal Sainrat) financée par Airbus (Benoît Triquet est notre correspondant technique).

Enfin, dans le cadre d'un projet avec Thales, nous avons acheté le multi-cœurs TMS320C6678 de Texas Instrument [Tex13] composé de 8 cœurs DSP reliés par un bus partagé. Son architecture est décrite dans la partie 4.3.1.

Gestion temporelle Ces quatre plates-formes présentent les mêmes particularités en mode *bare-metal* : chaque cœur possède des compteurs de cycles locaux synchrones mais comme les cœurs ne démarrent pas au même moment, cela implique que leurs horloges locales, bien que synchrones, ont des dates de réveil (ou offsets) non identiques et non prédictibles. De plus, l'accès à l'horloge globale (si elle existe) a un coût élevé et un délai d'accès variable. Nous devons alors définir pour chacune des plates-formes un protocole de synchronisation le plus précis possible pour ensuite exécuter les séquenceurs temps réel. Deux exemples de barrières de synchronisation sont décrites pour le SCC et le TMS dans la partie 4.3.2, la première avec une précision de $4\mu s$ et la deuxième de 70ns. Dans notre programmation en ZOL sur le TILERA, nous avons obtenu une précision pire cas de $0.5\mu s$ mais une observation moyenne de 60ns.

1. `https://communities.intel.com/community/marc`

Gestion des communications L'avantage des quatre plates-formes est également de gérer explicitement les communications. Pour le Tms, cela vient du placement en mémoire et des moments d'accès. Pour les pluri-cœurs, l'idée est de développer des librairies de passage de messages et de routage efficaces sur les réseaux sur puce.

1.2.4 Intégration dans un système à criticité mixte

La puissance de calcul offerte par les multi/pluri-cœurs entraîne également une plus forte intégration d'applications sur ces plates-formes, c'est-à-dire qu'il faut partager les ressources entre applications de caractéristiques différentes et surtout de niveaux de criticité divers. On retrouve le concept d'avionique modulaire intégrée déjà mentionnée précédemment. Dans la communauté temps réel, on parle de systèmes à « *criticité mixte* », c'est-à-dire des ensembles de tâches avec des contraintes de respect d'échéances temporelles variables. Les tâches de haute criticité doivent absolument respecter leurs échéances tandis que les autres peuvent en manquer.

Nous avons commencé des travaux avec mes collègues du projet Toast (Olivier Baldellon, Matthieu Roy - LAAS et Christine Rochange - IRIT), mes collègues du projet européen FP7 Dreams, en particulier ceux de Thales (Madeleine Faugère, Sylvain Girbal et Daniel Gracia Pérez), et notre post-doctorante RTRA Angeliki Kritikakou, afin d'offrir un environnement d'exécution prédictible en présence de tâches moins critiques. Dans sa version actuelle, notre méthodologie considère (1) une plate-forme de type multi-cœurs ; (2) deux types de tâches (hautement critiques ou non critiques) ; et assure le respect des échéances des tâches critiques grâce à un contrôleur en-ligne qui surveille leur comportement. Dès que celles-ci présentent un risque de dépassement d'échéance, du fait des accès trop fréquents des tâches moins critiques aux ressources partagées, ces dernières sont interrompues puis reprises lorsque les tâches critiques ont retrouvé des performances correctes. La méthode est sûre pour les tâches critiques car elle repose sur une analyse statique de calcul de WCET (avec l'outil Otawa [BCRS10] développé à l'IRIT) et augmente les performances des tâches moins critiques.

1.3 Organisation du document

Les travaux connexes sont décrits dans chaque chapitre. Le document est organisé en quatre chapitres techniques ; suivis d'un chapitre de bilan et perspectives (chapitre 6) ; puis d'un CV détaillé (chapitre 7).

Le chapitre 2 est consacré au langage Prelude. Nous décrivons la syntaxe, la sémantique et la génération de code multi-tâches. Nous détaillons ensuite le développement d'une étude de cas avionique open source, Rosace, réalisée avec David Saussié. Le chapitre se termine avec la présentation des extensions de Prelude proposées par Rémy Wyss afin de faciliter le travail de spécification du concepteur.

Le chapitre 3 présente les principes à appliquer sur une cible multi/pluri-cœurs pour obtenir un niveau suffisant de prédictibilité. La première partie décrit le socle de nos travaux avec les idées fondatrices de l'environnement SchedMCore. Les parties suivantes présentent les modèles d'exécution définis à l'ONERA. Leur validation, avantages et inconvénients sont discutés.

Le chapitre 4 se concentre sur l'aspect implantation effective des programmes sur la cible. La première partie décrit les méthodes de calcul de placement sur les cibles à base de résolution de contraintes. Cela suppose de connaître le modèle d'exécution et l'application. La dernière partie est dédiée à la description des exécutifs *bare-metal* réalisés sur le SCC et le Tms.

Le dernier chapitre technique, chapitre 5, présente les travaux sur l'exécution prédictible dans un cadre à criticité mixte.

Les publications me concernant mentionnées dans les pages suivantes et illustrant les différents axes de recherche sont listées dans la partie 7.5 p. 95, tandis que la bibliographie générale se trouve p. 101.

Chapitre 2

Prelude : langage de spécification de systèmes multi-périodiques

L'objectif de ce chapitre est de présenter le langage d'assemblage Prelude, d'illustrer son utilisation sur plusieurs études de cas, d'expliquer les principes de génération de code multi-tâches et de décrire les extensions en cours de développement.

2.1 Contexte

Les langages synchrones [BCE03 ; And05] sont utilisés avec succès depuis de nombreuses années pour spécifier formellement les systèmes de type contrôle/commande car ils offrent l'avantage à la fois de fournir un bon niveau d'abstraction et de s'appuyer sur des générateurs de code exécutable éprouvés. Les compilateurs génèrent un code séquentiel, ce qui donne lieu à des implantations mono-tâches.

2.1.1 Besoins et objectifs

Au cours de plusieurs contrats industriels, nous avons constaté que les applications de contrôle/commande manipulaient de grands nombres de blocs simples (entre 500 et 5000) qui devaient ensuite être assemblés. Du fait de la taille de ces blocs et des contraintes temporelles issues de l'automatique, ils ne pouvaient pas tous être exécutés à la fréquence la plus rapide. Le résultat est que les contrôleurs sont définis comme des assemblages de blocs avec des périodes différentes. Le langage Lustre n'a pas été conçu pour ce genre de spécification, ne s'y prête pas vraiment bien et n'est donc pas utilisé à cet effet par les industriels. A la place, des solutions « *maison* » ont été mises en place : (1) un intégrateur a en charge de combiner les blocs de façon à respecter les contraintes, (2) des langages ad hoc sans sémantique formelle décrivent partiellement les contraintes entre blocs et des séquences sont calculées automatiquement avec des outils dédiés.

Ce constat nous a conduit à réfléchir une méthode d'aide à la conception d'assemblages multi-périodiques. Du fait du succès de Lustre et des suites d'outil associées, des connaissances acquises par les industriels sur la programmation synchrone, il nous a paru pertinent de fournir une approche similaire à celle utilisée pour les blocs élémentaires : définir un langage formel avec des bases synchrones, proposer des règles de compilation simples qui pourraient être certifiées ultérieurement puisque proches de celles Scade, préparer l'exécution parallèle. Les travaux de thèse de Julien Forget, financés par EADS Astrium (2006-2009), ont permis d'étendre l'utilisation des langages synchrones à cette étape d'intégration en montrant qu'il était possible de spécifier un système multi-périodique à un niveau abstrait synchrone, puis de générer automatiquement un système multi-tâches. Ce système peut ensuite être exécuté par des ordonnanceurs standards et respecte la sémantique initiale du programme quelle que soit la politique d'ordonnancement choisie à l'implantation. Depuis deux ans, l'utilisabilité et l'expressivité de Prelude sont évaluées avec des ingénieurs / chercheurs en automatique dans le cadre d'un projet interne PR Forces 3 (2013-2015) et d'une collaboration avec David Saussié (Polytech Montréal). Ces échanges nous ont permis de valider les choix réalisés dans le langage et d'améliorer l'interface utilisateur (ex. clarification des messages d'erreur du compilateur, génération automatique de squelettes des nœuds importés, interfaçage avec nœuds importés écrits en Lustre).

2.1.2 Travaux connexes

Protocole de communication. Tant que le code généré à partir d'un programme synchrone est séquentiel, la gestion du flot de données et des valeurs utilisées est relativement simple. Lorsqu'on accepte une exécution multi-tâches, les différentes tâches peuvent s'entrelacer et il est impératif de fournir un mécanisme supplémentaire afin de conserver la sémantique initiale. Sofronis et al. [STC06 ; CSST08] ont défini un protocole de communication appelé *Dynamic Buffering Protocol* (DBP) assurant la préservation du comportement de tâches exprimées en Lustre lorsqu'elles sont ordonnancées par des politiques RM (*rate monotonic*) ou EDF (*earliest deadline first*). Ce protocole est à la base de toutes les extensions synchrones permettant l'exécution de code multi-tâches. Il a été étendu et appliqué pour Matlab/Simulink par [DWS09].

Langages de description d'architecture fonctionnelle

Avant et durant la thèse de Julien Forget. Une bibliographie détaillée se trouve dans le manuscrit de thèse de J. Forget [For09].

Matlab/Simulink [The14] est un langage de modélisation haut-niveau, développé par The Mathworks, largement utilisé dans de nombreux domaines d'application industriels. Les systèmes considérés doivent être harmoniques, c'est-à-dire que les périodes sont multiples les unes des autres (ce qui est plus restrictif que multi-périodique). Le langage propose des opérateurs de changement de période relativement proches de ceux de Prelude. Il est possible de générer, à partir de l'outil Real-time Workshop, du code pour des ordonnanceurs de type RM dont les communications se font par sémaphore.

Le langage Clara est dédié à la description de l'architecture fonctionnelle de systèmes réactifs. L'environnement proposé [Dur98 ; FDT04] part d'une modélisation formelle à l'aide de Clara et fournit des techniques de vérification formelle basées sur les réseaux de Petri temporisés. Prelude réutilise certains concepts de Clara mais avec une approche synchrone.

De nombreuses extensions ont déjà été proposées dans le monde synchrone pour traiter de plus larges familles de systèmes. [SL97] ont proposé de combiner des programmes en relation affine et [MTGL08] a défini une restriction de ces relations afin de spécifier des systèmes multi-périodiques. Ces travaux portent principalement sur la vérification formelle. [GN03 ; GNP06] définissent une distribution automatique de programmes Lustre. Le programmeur doit classer les horloges en partitions, et un programme Lustre est ensuite généré par partition. Adrian Curic [Cur05] est le premier à avoir explicitement introduit les aspects temps réel dans un langage synchrone. Des *hypothèses* permettent de spécifier le rythme de base d'un programme (la durée temps réel d'un instant) ainsi que les durées d'exécution des nœuds. Des *exigences* permettent de contraindre la date à laquelle un flot doit être produit ainsi que la latence maximale entre deux flots. Des horloges périodiques sont alors définies, elles valent vrai une fois toutes les n itérations de base. La compilation s'applique à des plates-formes d'exécution de type TTA (Time-Triggered Architecture) [KB03]. [ACGR09] a étendu les résultats de Curic en introduisant des opérateurs spécifiant des conditions d'activation périodiques et des primitives permettant de gérer plusieurs modes d'exécution. La compilation plus simple et plus générique produit un ensemble de tâches temps réel concurrentes. Les communications inter-tâches sont gérées par le protocole de communication de [STC06]. Le calcul des priorités des tâches n'est en revanche pas détaillé. [CGP08] ont exprimé des systèmes multi-périodiques en Scade mais la suite d'outil n'a pas ensuite été étendue pour générer le code ad hoc.

Depuis la thèse de Julien Forget. L'importance de définir des langages formels pour décrire des ensembles de tâches dépendantes communicantes a été montrée dans [Bar12]. Les auteurs de [OTE12] ont développé une librairie Modelica permettant de spécifier des systèmes multi-périodiques à l'aide des opérateurs et la sémantique de Prelude. L'auteur de [Kra14] a également intégré les opérateurs et de la sémantique de Prelude dans son langage polychrone.

Les auteurs de [BT13] se sont intéressés à la compilation multi-tâches d'assemblages multi-périodiques exprimés en relation affine, repartant ainsi des travaux de [SL97]. Ils ont également proposé des moyens d'analyses d'ordonnançabilité à base d'EDF efficaces. Les auteurs de [BJF12] ont défini une traduction de programmes Scade en une implantation Oasis (Oasis est un système d'exploitation prédictible développé pour le domaine nucléaire). La façon d'exprimer les relations multi-périodiques est néanmoins laissée de côté.

Haibo Zeng et Marco Di Natale ont mené de nombreuses recherches sur la problématique d'exprimer des systèmes multi-périodiques dans un paradigme synchrone et de générer des ensembles de tâches ordonnançables [DGZS10 ; ZD12 ; AZDG13]. [YKRB14] ont étudié l'ordonnançabilité de tâches issues de

langages tels que PRELUDE dans un cadre à criticité mixte. Leur langage de départ est du C annoté [YRBG13].

2.2 Présentation du langage d'assemblage

Un programme PRELUDE décrit l'assemblage d'un ensemble de nœuds importés considérés comme des boîtes noires. Il s'agit de nœuds écrits dans un langage de programmation indépendant de PRELUDE. Un programme PRELUDE spécifie de manière modulaire les relations entre les entrées et sorties de nœuds importés, ainsi que leurs liens avec les entrées / sorties du nœud principal. L'environnement SCHEDMCORE, présenté dans la section 3.2 et développé dans la thèse de Mikel Cordovilla, permet entre autre de simuler fonctionnellement des assemblages PRELUDE dont les nœuds importés sont en C.

2.2.1 Description de la syntaxe PRELUDE

Le langage d'assemblage PRELUDE a été initialement défini dans [FBLP08] puis complété dans [For09] et reprend de nombreuses idées du langage LUSTRE. LUSTRE [CPHP87] est un langage synchrone flot de données développé à Verimag. Pour trouver une description complète et détaillée du langage, le lecteur pourra se référer à [Hal92]. De manière informelle, un programme LUSTRE exprime des relations entre les variables d'entrée et de sortie, chaque variable étant une suite infinie de valeurs appelée *flot*. Selon l'hypothèse *synchrone*, un système est décrit sur une échelle de temps discret global et toutes les transformations de flots sont supposées se faire durant une unité de temps appelée *instant*. Cette dernière hypothèse étant trop restrictive pour une implantation multi-tâches, PRELUDE se base sur l'hypothèse *synchrone relâchée* d'A. Curic [Cur05] qui stipule que chaque calcul doit terminer avant sa prochaine activation (et non dans l'instant logique). Cela relâche ainsi les contraintes d'échéance d'exécution des nœuds importés et rapproche les spécifications vers celles habituelles du domaine de l'ordonnancement.

Un programme PRELUDE manipule des flots construits à partir de constantes, de variables, de n-uplets de valeurs, d'appels de nœuds importés appliqués sur des flots et de combinaison d'opérateurs d'assemblage. Un flot est une suite de paires $(v_i, t_i)_{i\in\mathbb{N}}$, où v_i est la valeur du flot dans son domaine $\mathcal{V}$ et $t_i \in \mathbb{N}$ est la date de son occurrence, on a $\forall i, t_i < t_{i+1}$. L'*horloge* d'un flot indique si le flot est présent ou non sur l'horloge de base, celle qui rythme tous les instants. L'horloge de base est exprimée sur une échelle de temps donnée sur les entiers naturels. Les variables manipulées par le langage sont des flots dont les horloges sont *strictement périodiques*. Une horloge est strictement périodique si l'intervalle entre deux activations successives est constant.

Définition 1 (Horloge strictement périodique)**.** *Une* horloge strictement périodique *est définie par une paire (p, q) où $p \in \mathbb{N}^\star$ est la période et $q \in \mathbb{Q}^+$ la phase (ou date d'activation) relative. Ainsi, si $h[i]$ représente la date de la ième valeur de l'horloge h, alors*

- *pour tout i, $h[i+1] - h[i]$ vaut toujours la même constante entière p ;*
- *$h[0] = p \times q$ est le premier instant d'activation.*

Pour tout flot x, on note par $\pi(x)(= p)$ sa période et par $\phi(x)(= pq)$ sa phase.

Ainsi, l'horloge la plus rapide est $(1, 0)$ car elle est présente à tout instant. De même $(2, 0)$ représente l'horloge activée toutes les 2 unités de temps.

Les opérateurs PRELUDE sont les suivants :

1. *cst* `fby` x est l'opérateur *followed by* défini en SCADE et LUCID SYNCHRONE. Cet opérateur permet d'initialiser un flot par la constante *cst* et de prendre la valeur retardée d'un instant du flot x ;
2. $x *\hat{}\, k$ est un opérateur d'accélération de rythme. Appliqué au flot x, cet opérateur permet de construire un flot qui va k fois plus rapidement (i.e., k fois plus fréquent) ;
3. $x/\hat{}\, k$ est un opérateur de décélération de rythme. Appliqué au flot x, cet opérateur permet de construire un flot qui va k fois plus lentement (i.e., k fois moins fréquent).
4. $x \sim> q$ est un opérateur de décalage de phase. Appliqué au flot x, il construit un flot retardé de $q * \pi(x)$ par rapport à la phase de x ;
5. `tail` x est un opérateur de suppression d'élément initial. Il ôte la première valeur de x et coïncide ensuite avec x ;
6. *cst* :: x est un opérateur de concaténation d'élément initial. Il produit la constante *cst* une période plus tôt que x et produit ensuite x ;

7. $\tau(x)$ est l'opérateur consistant à appliquer le nœud importé τ au flot x.

Voici une illustration des comportements élémentaires des opérateurs PRELUDE :

	1	2	3	4	5	6	7	horloge associée
x	$x[1]$		$x[2]$		$x[3]$		$x[4]$	(2,0)
$\tau(x)$	$\tau(x[1])$		$\tau(x[2])$		$\tau(x[3])$		$\tau(x[4])$	(2,0)
0 `fby` x	0		$x[1]$		$x[2]$		$x[3]$	(2,0)
x $*\hat{}$ 2	$x[1]$	$x[1]$	$x[2]$	$x[2]$	$x[3]$	$x[3]$	$x[4]$	(1,0)
x $/\hat{}$ 2	$x[1]$				$x[3]$			(4,0)
x $\sim>$ 1/2		$x[1]$		$x[2]$		$x[3]$		(2,1/2)
`tail` x			$x[2]$		$x[3]$		$x[4]$	(2,1)
0 :: (`tail` x)	0		$x[2]$		$x[3]$		$x[4]$	(2,0)

Dans cet exemple, on part d'un flot x ayant une horloge de $(2,0)$ et qui prend des suites de valeurs $x[i]$. Dans la deuxième ligne, on applique un nœud importé τ sur x et le flot obtenu est *synchrone* avec x (deux flots sont synchrones s'ils ont la même horloge). `fby` ne modifie pas l'horloge et décale les valeurs d'une période. On constate que les autres opérateurs ont des actions directes sur les horloges : $\sim> q$, `tail` et :: modifient la phase ; $*\hat{}\,k$ multiplie la fréquence de l'horloge par k (et donc divise d'autant sa période) et $/\hat{}\,k$ divise sa fréquence par k (et donc multiplie d'autant sa période).

Exemple 1. *Considérons à présent un premier exemple complet assemblant 3 nœuds importés. La première étape consiste à lister les nœuds importés :*

```
imported node tau_1 (i:int) returns (o:int) wcet 1;
imported node tau_2 (i1, i2:int) returns (o:int) wcet 1;
imported node tau_3 (i1, i2:int) returns (o:int) wcet 1;
```

La signature pour chaque nœud est : type des entrées et des sorties, suivi d'une valeur de WCET (worst case execution time ou pire temps d'exécution) dont l'unité est la même que celle de la période. Ensuite, chaque entrée de l'assemblage est déclarée comme un capteur et chaque sortie comme un actionneur. Chacun doit également être accompagné d'une valeur de WCET.

```
sensor i1 wcet 1;
sensor i2 wcet 1;
sensor i3 wcet 1;

actuator o1 wcet 1;
actuator o2 wcet 1;
actuator o3 wcet 1;
```

Les entrées/sorties du programme principal sont considérées comme des activités réelles liées aux capteurs et actionneurs. A titre d'exemple, lire une entrée peut nécessiter d'accéder à un périphérique ce qui prend du temps et explique un WCET non nul. On peut toutefois associer un WCET nul à une entrée ou à une sortie, si on suppose que les données sont simplement en mémoire et que les activités d'envoi/réception sur les périphériques sont réalisées par des tâches extérieures au programme PRELUDE. *Enfin, on exprime les règles d'assemblage dans un programme* PRELUDE. *A noter que les rythmes des entrées doivent être spécifiées, tandis que ceux des sorties peuvent être inférés par le compilateur :*

```
node ex (i1 : rate(8,0); i2 : rate(20,0); i3 : rate(28,0))
returns (o1, o2, o3)
let
  o1 = tau_1((0 fby (0 fby i1))/^3);
  o2 = tau_2(i2*^5, i3*^7);
  o3 = tau_3(o1 ~>1/2 , (o2/^6) ~> 1/2);
tel
```

La syntaxe est semblable à du LUSTRE. *Un nœud est décrit par son nom, les variables d'entrée (ici, il y en a 3 i_1 à i_3) et les variables de sortie (également 3 variables o_1 à o_3). Après le mot clé* `let`, *on trouve l'ensemble des équations reliant les entrées/sorties du nœud.*

L'horloge des nœuds importés est : $\tau_1 = (24,0)$, $\tau_2 = (4,0)$, $\tau_3 = (24,12)$. Un comportement de ce programme est schématisé ci-dessous. On note $o[p]$ la p-ième valeur de o. Le pas de temps logique est d'une unité (ou top) mais les flots sont cadencés sur des horloges plus lentes et on ne connaît le comportement que sur des intervalles de temps et non le comportement interne à un intervalle. Cet aspect est similaire à la notion de temps logique en LUSTRE, *on suppose que toutes les équations sont résolues simultanément et le comportement détaillé à l'intérieur d'un instant n'a pas de sens. i_1 a une horloge de $(8,0)$ donc*

sa valeur est calculée dans chacun des intervalles $[8k, 8k+8[$. C'est également le cas pour 0 `fby` i_1 *et* 0 `fby` $(0$ `fby` $i_1)$*. Le flot o_1 est évalué sur l'intervalle $[24k, 24k+24[$. L'horloge la plus rapide de 4 tops est celle d'o_2. Entre $[0,4[$, on doit exécuter $i_2[1]$ (coût = 1 unité de temps) et $i_3[1]$ (coût = 1 unité de temps) (ces deux premières exécutions peuvent se faire en parallèle), puis $\tau_2(i_2[1], i_3[1])$ (coût = 1 unité de temps) et enfin $o_1[1]$ (coût = 1 unité de temps). On constate $i_2[1]$ doit être calculée dans $[0,4[$ car cette valeur est consommée par un nœud dont l'horloge $(4,0)$ est plus rapide que celle de i_2 de $(20,0)$. Toutes les combinaisons respectant les échéances sont acceptables.*

<table>
<tr><th>temps</th><th>$[0,4[$</th><th>$[4,8[$</th><th>$[8,12[$</th><th>$[12,16[$</th><th>$[16,20[$</th><th>$[20,24[$</th><th>$[24,28[$</th></tr>
<tr><td>$i1$</td><td colspan="2">$i_1[1]$</td><td colspan="2">$i_1[2]$</td><td colspan="2">$i_1[3]$</td><td>$i_1[4]$</td></tr>
<tr><td>0 fby i_1</td><td colspan="2">0</td><td colspan="2">$i_1[1]$</td><td colspan="2">$i_1[2]$</td><td>$i_1[3]$</td></tr>
<tr><td>0 fby (0 fby i_1)</td><td colspan="2">0</td><td colspan="2">0</td><td colspan="2">$i_1[1]$</td><td>$i_1[2]$</td></tr>
<tr><td>$o1$</td><td colspan="6">$\tau_1(0)$</td><td></td></tr>
<tr><td>i_2</td><td colspan="5">$i_2[1]$</td><td colspan="2">$i_2[2]$</td></tr>
<tr><td>i_3</td><td colspan="7">$i_3[1]$</td></tr>
<tr><td>o_2</td><td>$\tau_2(i_2[1], i_3[1])$</td><td>$\tau_2(i_2[1], i_3[1])$</td><td>$\tau_2(i_2[1], i_3[1])$</td><td>$\tau_2(i_2[1], i_3[1])$</td><td>$\tau_2(i_2[1], i_3[1])$</td><td>$\tau_2(i_2[2], i_3[1])$</td><td>$\tau_2(i_2[2], i_3[1])$</td></tr>
<tr><td>o_3</td><td colspan="3"></td><td colspan="4">$\tau_3(o_1[1], o_2[1])$</td></tr>
</table>

<table>
<tr><th></th><th>$[28,32[$</th><th>$[32,36[$</th><th>$[36,40[$</th><th>$[40,44[$</th><th>$[44,48[$</th><th>$[48,52[$</th><th>...</th></tr>
<tr><td>i_1</td><td></td><td colspan="2">$i_1[5]$</td><td colspan="2">$i_1[6]$</td><td>$i_1[7]$</td><td>...</td></tr>
<tr><td>0 fby i_1</td><td></td><td colspan="2">$i_1[4]$</td><td colspan="2">$i_1[5]$</td><td>$i_1[6]$</td><td>...</td></tr>
<tr><td>0 fby (0 fby i_1)</td><td></td><td colspan="2">$i_1[3]$</td><td colspan="2">$i_1[4]$</td><td>$i_1[5]$</td><td>...</td></tr>
<tr><td>o_1</td><td colspan="5">$\tau_1(i_1[2])$</td><td>$\tau_1(i_1[5])$</td><td>...</td></tr>
<tr><td>i_2</td><td colspan="3"></td><td colspan="3">$i_2[3]$</td><td>...</td></tr>
<tr><td>i_3</td><td colspan="6">$i_3[2]$</td><td>...</td></tr>
<tr><td>o_2</td><td>$\tau_2(i_2[2], i_3[2])$</td><td>$\tau_2(i_2[2], i_3[2])$</td><td>$\tau_2(i_2[2], i_3[2])$</td><td>$\tau_2(i_2[3], i_3[2])$</td><td>$\tau_2(i_2[3], i_3[2])$</td><td>$\tau_2(i_2[3], i_3[2])$</td><td>...</td></tr>
<tr><td>o_3</td><td colspan="2"></td><td colspan="4">$\tau_3(o_1[2], o_2[7])$</td><td>...</td></tr>
</table>

Codage de l'exemple des commandes de vol en PRELUDE

Reprenons l'exemple des commandes de vol simplifiées de la figure 1.1 p. 2 décrit dans l'introduction. Les nœuds importés sont les fonctions décrites dans les boîtes comme la loi de guidage ou la loi de pilotage. Les noms des nœuds sont ceux en abrégé. Ainsi, la spécification de l'assemblage commence par la liste des nœuds importés.

```
imported node FA(i: real) returns (o: real)  wcet 5;
imported node LA(i1, i2: real) returns (o: real) wcet 5;
...
```

On spécifie pour chaque nœud le type des entrées et des sorties. Ainsi, le nœud F_A prend une entrée réelle et renvoie une sortie réelle. On suppose que les nœuds importés sont synchrones et polymorphes du point de vue des horloges. Ainsi, F_A prend une entrée dont l'horloge est de type polymorphe α (c'est-à-dire qu'elle sera de la forme (k,p) avec $k \in \mathbb{N}$, $p \in \mathbb{Q}$ et $pq \in \mathbb{N}$) et renvoie une sortie dont l'horloge est de même type α. On précise également le WCET de chaque nœud.

On précise ensuite les entrées / sorties du nœud principal et on suppose ici que les données se trouvent en mémoire (ce qui implique que les WCET sont nuls).

```
sensor angle wcet = 0;
```

On décrit ensuite l'assemblage. Plusieurs découpages modulaires sont possibles et dans l'exemple ci-dessous, on a choisi d'écrire un nœud unique pour programmer le système.

```
node cdv (angle, acc, pos : rate (30, 0);  pos_c: rate (70, 0))
returns (ordre)
var acc_c, angle_c, pos_i, acc_i,  angle_o,  acc_o, pos_o;
let
  acc_i = AC(acc);
  pos_i = AP(pos);
  angle_o = FA(angle);
  acc_o = FP(acc_i*^3/^4);
  pos_o = FG(pos_i*^3/^7);
  ordre = LA(angle_o,  (angle_c*^4)/^3);
```

```
  angle_c = LP (acc_o,((0 fby acc_c)*^7)/^4);
  acc_c = LG(pos_o,pos_c);
tel
```

Dans cet exemple, sept variables internes sont utilisées pour des calculs intermédiaires. Les premières équations produisent des variables à des rythmes synchrones. L'ordre imposé par les lois d'asservissement utilise la donnée générée par *FA* et la donnée `angle_c` produite à 40ms. Pour être utilisable par un nœud à 30, il faut resynchroniser cette donnée. Le choix fait ici est d'accélérer par 4 puis de ralentir par 3. De la même manière, la variable `acc_c` est produite toutes les 70 et est consommée à rythme de 40. Le choix est de prendre les valeurs avec délai `0 fby acc_c` puis d'accélérer par 7 et de ralentir par 4. Cette donnée est moins critique qu'`angle_c` et peut être utilisée avec une valeur plus ancienne.

2.2.2 Sémantique

Un programme est correct si les opérations appliquées respectent les règles de typage des données et des horloges. Il faut en particulier vérifier qu'aucune équation n'essaie de combiner deux flots qui ne sont pas présents aux mêmes instants. Dans cette section, on rappelle la sémantique d'un programme par trace puis par la notion de mots de dépendance de données. Cette dernière notion contient toutes les informations relatives au programme, aussi bien les horloges que les consommations effectives de données.

Sémantique de Kahn

La sémantique de PRELUDE peut être exprimée grâce à une sémantique de Kahn sur les flots [Kah74] et à une adaptation de la sémantique synchrone présentée dans [CP03] au *Tagged Signal Model* [LS96]. Pour chaque opération *ops*, $ops^{\#}(s_1, \ldots, s_n) = s'$ signifie que l'opération *ops* appliquée au n-uplet $(s_1, \ldots, s_n)$ produit la séquence s'. Le terme $(v, t).s$ représente le flot dont la tête a la valeur v et le *tag* t et dont la queue est la séquence s. La sémantique des opérations est définie de manière inductive sur la structure des arguments. La sémantique des opérations PRELUDE, issue de [For09], est donnée en figure 2.1.

$$\begin{array}{lcll}
\texttt{fby}^{\#}(v, (v', t).s) & = & (v, t).\texttt{fby}^{\#}(v', s) & \\
\tau^{\#}((v, t).s) & = & (\tau(v), t).\tau^{\#}(s) & \text{où } \tau \text{ est un nœud importé} \\
\hat{}^{\#}((v, t).s, k) & = & \prod_{i=1}^{k}(v, t'_i).\hat{}^{\#}(s, k) & \text{où } t'_1 = t \text{ et } t'_{i+1} - t'_i = \pi(s)/k \\
/\hat{}^{\#}((v, t).s, k) & = & \begin{cases} (v, t)./\hat{}^{\#}(s, k) & \\ /\hat{}^{\#}(s, k) & \end{cases} & \begin{array}{l}\text{si } k * \pi(s) | t \\ \text{sinon}\end{array} \\
\sim>^{\#}((v, t).s, q) & = & (v, t'). \sim>^{\#}(s, q) & \text{avec } t' = t + q * \pi(s) \\
\texttt{tail}^{\#}((v, t).s) & = & s & \\
::^{\#}(cst, s) & = & (cst, \phi(s) - \pi(s)).s &
\end{array}$$

FIGURE 2.1 – Sémantique de Kahn

Mots de dépendance de données

Un programme PRELUDE peut être vu comme un *graphe de précédences étendues* (V, E) où l'ensemble de nœuds V correspond aux fonctions importées ainsi qu'aux variables d'entrée et de sortie du système global ; et où les transitions $E \subseteq V \times V$ sont étiquetées par des listes d'opérateurs appliqués entre deux tâches. Le graphe de précédences étendues pour les commandes de vol est donné dans la figure 2.2.

La sémantique du langage d'assemblage implique un ordre partiel dans l'exécution des tâches ainsi qu'un schéma précis des consommations de données. En effet, si un nœud importé A est appliqué sur un flot résultat d'un autre nœud importé B alors le nœud B doit s'exécuter avant le nœud A. Le langage permet n'importe quelle combinaison, mais nous supposerons dans la suite que les délais ne peuvent être appliqués qu'au début, c'est-à-dire la combinaison ne peut être que de la forme $\texttt{fby}^k$, combinaison des autres opérateurs. Les *mots de dépendance de données*, introduits dans [BCF09 ; PFB11] puis étendus dans [WBPF13], sont un moyen concis de représenter les dépendances entre les flots. Plus précisément,

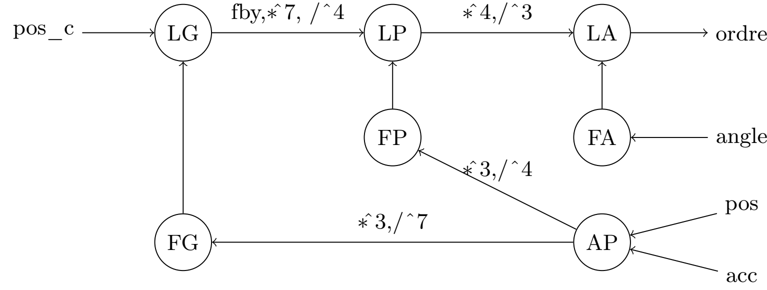


FIGURE 2.2 – Graphe de précédences étendues des commandes de vol

un mot de dépendance de données décrit les valeurs de i qui sont nécessaires pour calculer les valeurs de o dans l'expression $o = ops(i)$, où $ops= op_1.op_2.\cdots.op_n$ représente la composition $op_n(\dots(op_2(op_1(i))))$ avec $op ::= \texttt{fby} \mid /\hat{}k \mid *\hat{}k \mid \sim\!> k \mid \texttt{tail} \mid ::$. On note $o[p]$ la p-ième valeur de o et par $o[p] \leftarrow i[n]$ le fait que nous utilisons $i[n]$ pour calculer $o[p]$.

Définition 2 (Mot de dépendance de données)**.** *Un mot de dépendance de données w est défini par la grammaire suivante :*

$$\begin{array}{lcl} w & ::= & (-1, d_0).(j, d).u \\ u & ::= & (j, d)|u.u \end{array}$$

avec $d_0 \in \mathbb{N}$, $j, d \in \mathbb{N}^+$. Si le mot $w = (-1, d_0)(j_1, d_1)(j_2, d_2)\dots(j_n, d_n)$ représente la dépendance de données entre les flots o et i, on a donc les dépendances suivantes, $\forall p \in \mathbb{N}^\star$:

$$o[p] \leftarrow \begin{cases} init & si\ p \in [1, d_0] \\ i[j_1] & si\ p \in [d_0+1, d_0+d_1] \\ i[j_1+j_2] & si\ p \in [d_0+d_1+1, d_0+d_1+d_2] \\ \dots & \\ i[q \cdot (\Sigma_{k\in[2,n]}\ j_k) + \Sigma_{k\leq l}\ j_k] & si\ q \in \mathbb{N}, l \in [1,n], p \in [a_p, b_p] \\ & avec\ a_p = q \cdot (\Sigma_{k\in[2,n]}\ d_k) + \Sigma_{k\leq l-1}\ d_k + 1 \\ & et\ b_p = q \cdot (\Sigma_{k\in[2,n]}\ d_k) + \Sigma_{k\leq l}\ d_k \end{cases}$$

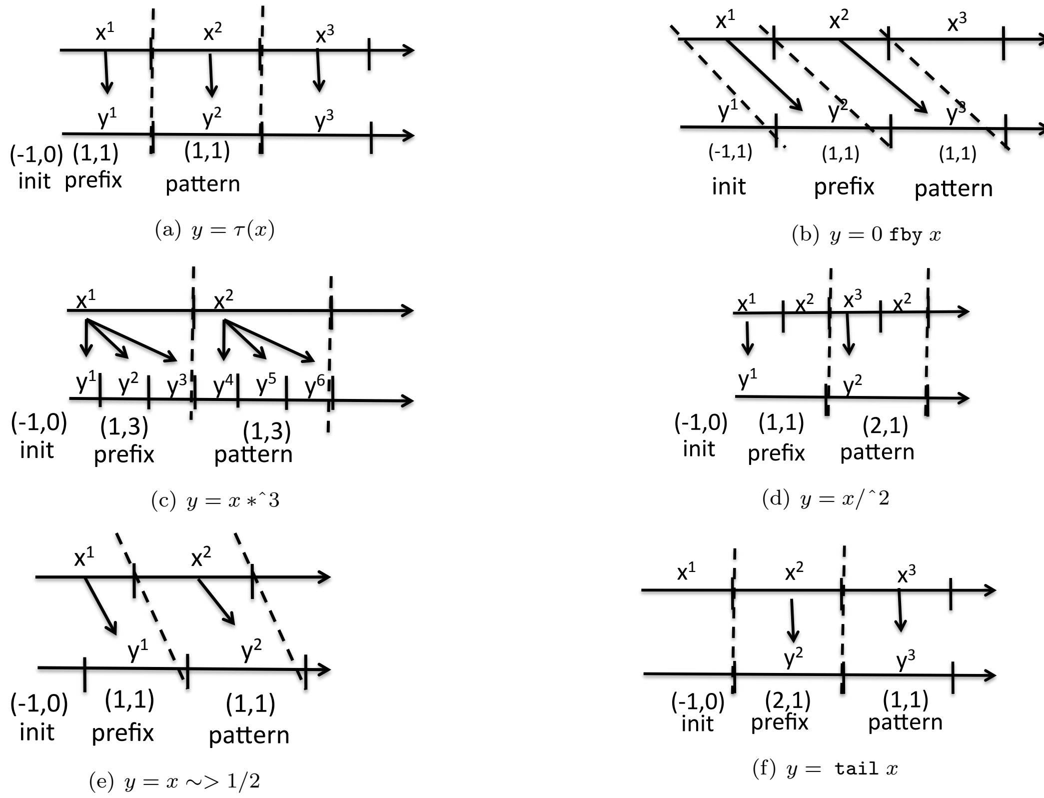
Les deux premières lettres $(-1, d_0)(j_1, d_1)$ correspondent au préfixe du mot à savoir : le nombre d_0 de valeurs d'initialisation consommées à cause des délais, la première instance j_1 à être réellement consommée et d_1 le nombre de fois où elle est consommée. Le reste du mot est le motif périodique de consommation se répétant à l'infini.

Exemple 2. $(-1, 0)(1, 1)(1, 1)$ *est le mot de dépendance de données le plus simple, on a la dépendance $o[p] \leftarrow i[p]$ pour tout $p \in \mathbb{N}^\star$. Par exemple dans l'expression* ordre = LA(FA(angle), (angle_c $*\hat{}4$)/ˆ3)*, le mot de dépendance pour* ordre ← angle *est* $(-1, 0)(1, 1)(1, 1)$*. La figure 2.3 montre les dépendances pour les opérateurs de base.*

Proposition 1. *Soit $o = ops(i)$, où $ops = \big(\texttt{fby}^{\,p_0},\ combinaison\ (*\hat{}\,k_i, /\hat{}\,k_i, \sim\!> q_i,\ \texttt{tail}\ , ::)\big)$. Les dépendances entre o et i peuvent être exprimées avec un mot de dépendance de données w dont la longueur est inférieure à $\prod_{/\hat{}k_i} k_i + 2$.*

w peut être calculé inductivement à partir des opérateurs apparaissant dans ops. On part du mot initial $(-1, 0)(1, 1)(1, 1)$ qui exprime que les deux flots sont en dépendance simple et qui correspond au cas ops= []. On note $l(w) = n - 1$ la longueur du motif de w (le préfixe de taille constante 2 n'est pas pris en compte dans l) et $nd(w) = \Sigma_{2\leq i\leq n}\ d_i$. Si le mot courant est $w = (-1, d_0)\ (j_1, d_1)\dots(j_n, d_n)$ et que le prochain opérateur est op, alors on calcule w' comme suit :

1. *si $op = \texttt{fby}$, alors $w' = (-1, d_0 + 1)(j_1, d_1)\dots(j_n, d_n)$;*
2. *si $op = *\hat{}\,k$, alors $w' = (-1, d_0 \times k)(j_1, d_1 \times k), \dots, (j_n, d_n \times k)$;*
3. *si $op = /\hat{}\,k$, alors le calcul est plus compliqué. On doit supprimer $k - 1$ consommations toutes les k dans w. On commence par le préfixe $(-1, \lceil d_0/k \rceil)(j'_1, d'_1)$. Si $d_0\ mod\ k = 0$ alors $j'_1 = j_1$ et $d'_1 = \lceil d_1/k \rceil$. Sinon $j'_1 = \Sigma_{l\leq p}\ j_l$ avec $p = min_m\{(d_0\ mod\ k + \Sigma_{l\leq m}\ d_l) \geq k\}$ et $d'_1 = \lceil (d_0 + \Sigma_{l\leq p}\ j_l - k)/k \rceil$. On doit ensuite copier r fois le sous-mot $(j_{p+1}, d_{p+1})\dots(j_{p+l(w)-1}, d_{p+l(w)-1})$*



(a) $y = \tau(x)$ (b) $y = 0$ `fby` x

(c) $y = x *\hat{}3$ (d) $y = x/\hat{}2$

(e) $y = x \sim> 1/2$ (f) $y =$ `tail` x

Figure 2.3 – Dépendances induites par les opérateurs de base

pour obtenir le mot w_i tel que $nd(w_i)$ soit divisible par k. On doit ensuite retenir une dépendance toutes les k fois du mot w_i. Cette opération produit w' dont la longueur est inférieure à $n \times (k+2)$.

4. *si op=$\sim>$ k, alors $w' = w$;*
5. *si op=* `tail` *, alors il y a trois cas :*
   (a) *si $d_0 > 0$, alors $w' = (-1, d_0 - 1)(j_1, d_1) \ldots (j_n, d_n)$*
   (b) *sinon, si $d_1 > 1$, alors $w' = (-1, 0)(j_1, d_1 - 1) \ldots (j_n, d_n)$*
   (c) *sinon $w' = (-1, 0)(j_1 + j_2, d_2)(j_3, d_3) \ldots (j_n, d_n)(j_2, d_2)$;*
6. *si op=::, alors $w' = (-1, d_0 + 1)(j_1, d_1) \ldots (j_n, d_n)$*

Exemple 3. *On illustre dans le tableau ci-dessous la construction des mots sur l'expression $o = ((dft$ `fby` $x) *\hat{}4)/\hat{}3$.*

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*mot*
x	$x[1]$				$x[2]$				$x[3]$		*(-1,0)(1,1)(1,1)*
$\xleftarrow{\texttt{fby}} x$	dft				$x[1]$				$x[2]$		*(-1,1)(1,1)(1,1)*
$\xleftarrow{\texttt{fby},*\hat{}4} x$	dft	dft	dft	dft	$x[1]$	$x[1]$	$x[1]$	$x[1]$	$x[2]$	$x[2]$	*(-1,4)(1,4)(1,4)*
$\xleftarrow{\texttt{fby},*\hat{}4,/\hat{}3} x$	dft			dft			$x[1]$			$x[2]$	*(-1,2)(1,1)(1,1)(1,2)(1,1)*

2.3 Génération de code multi-tâches

Lorsque le programme est correct, le compilateur extrait l'ensemble de tâches dépendantes temps réel qui communiquent de manière déterministe à l'aide d'un protocole de communication. A l'intégration sur la cible, il faut également faire le lien entre l'horloge globale sur $\mathbb{N}$ définie dans le langage et le temps physique du système. Nous décrivons dans cette section (1) l'ensemble de tâches, (2) les contraintes de précédence étendue et (3) le protocole de communication générés.

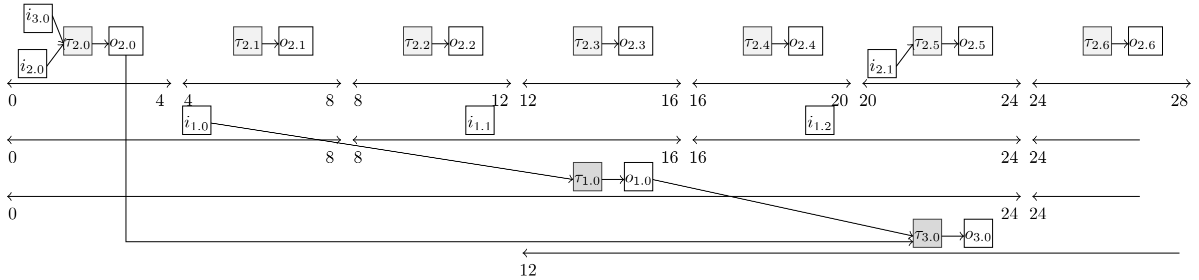
2.3.1 Ensemble de tâches temps réel

A partir d'un assemblage décrit en PRELUDE, le compilateur produit un système de tâches temps réel qui consiste en l'ensemble des nœuds importés. Concernant les caractéristiques temps réel : le WCET est spécifié dans le programme lors de la déclaration d'importation du nœud, la période et la date d'activation sont automatiquement calculées par le calcul d'horloge décrit dans [FBLP08]. Dans le langage présenté dans ce document, il n'est pas fait mention de l'opérateur `due to` qui permet de jouer sur l'échéance.

Modèle de tâches temps réel. Le modèle de tâches considéré est celui issu des travaux de Liu et Layland [LL73]. Une tâche τ est définie par sa date d'activation ou de réveil (O), sa période (T), son échéance relative (D) et sa durée d'exécution maximale (C), i.e. τ=(O,T,D,C). On note $\tau_{i.j}$ la j-ième instance de τ_i. La deadline absolue, notée $d_{i.j}$, de l'instance $\tau_{i.j}$ est alors $jT_i + D_i$.
A noter : les instances des tâches commencent à 0 tandis que les numéros des données commencent à 1.

L'extraction des tâches temps réel à partir d'un programme PRELUDE est détaillée dans [FBLP10; FBLP09].

Exemple 4. *A partir de l'exemple 1 p. 12, on obtient neuf tâches :* $\tau_1 = (0, 24, 24, 1)$*,* $\tau_2 = (0, 4, 4, 1)$*,* $\tau_3 = (12, 24, 24, 1)$*,* sensor_$i_1 = (0, 8, 8, 1)$*,* sensor_$i_2 = (0, 20, 20, 1)$*,* sensor_$i_3 = (0, 28, 28, 1)$*,* actuator_$o_1 = (0, 24, 24, 1)$*,* actuator_$o_2 = (0, 4, 4, 1)$*,* actuator_$o_3 = (12, 24, 24, 1)$*. Le comportement temporel des tâches est décrit ci-dessous (les flèches représentent les contraintes de précédence entre instances de tâches) :*



Dans l'exemple des commandes de vol, on obtient les sept tâches $L_A = (0, 10, 10, 1)$*,* $F_A = (0, 10, 10, 1)$*,* $A_P = (0, 10, 10, 2)$*,* $F_P = (0, 20, 20, 3)$*,* $L_P = (0, 20, 20, 3)$*,* $F_G = (0, 40, 40, 5)$ *et* $L_G = (0, 40, 40, 5)$*. Les entrées / sorties ont un WCET nul et donc n'apparaissent pas comme des tâches.*

Précédences étendues entre tâches. La sémantique du langage d'assemblage implique un ordre partiel dans l'exécution des tâches. En effet, si un nœud importé τ_2 est appliqué sur un flot résultat d'un autre nœud importé τ_1 alors le nœud τ_1 doit s'exécuter avant le nœud τ_2. Cela se traduit au niveau temps réel par une relation de précédence étendue. Le langage génère deux types de précédence :

- précédence indirecte : lorsqu'il y a utilisation d'un `fby` ou $\sim>$ q avec $q \geq 1$. En effet, dans ces deux cas de figure, le consommateur se réveille nécessairement après la période du producteur, la donnée produite est donc toujours disponible par définition.
- précédence directe : sinon, on n'est pas assuré par définition des flots que le producteur aura terminé son exécution avant le consommateur. Il faut donc que l'implantation force certaines instances de la tâche productrice à se terminer avant le début de certaines instances du consommateur. On parle alors de *contraintes de précédence étendue*, qui lient des sous-ensembles d'instances de tâches communicantes.

On note $\tau_{i.n} \rightarrow \tau_{j.n'}$ pour représenter une contrainte de précédence entre l'instance n de τ_i et l'instance n' de τ_j. On note $\mathcal{I}_n$ l'ensemble des entiers de l'intervalle $[0, n[$. La formalisation des précédences étendues est présentée dans [FBLP10; FBG10].

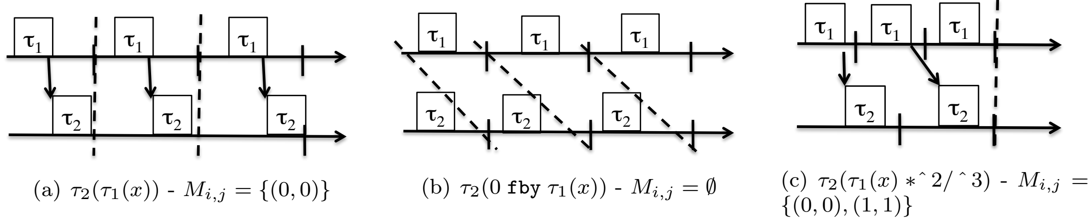
Définition 3 (Contraintes de précédence périodique étendue)**.** *On considère deux tâches* τ_i *et* τ_j*, leur plus petit commun multiple* $p = ppcm(T_i, T_j)$*, un entier strictement positif* $L_{i,j} \in \mathbb{N}^+$ *et un ensemble* $M_{i,j} \subseteq \mathcal{I}_{p \cdot L_{i,j}/T_i} \times \mathcal{I}_{p \cdot L_{i,j}/T_j}$ *(à noter que* $M_{i,j}$ *est toujours fini). La contrainte de précédence périodique étendue* $\tau_i \xrightarrow{M_{i,j}, L_{i,j}} \tau_j$ *est définie comme l'ensemble de contraintes de précédence entre les instances de tâches :*

$$\forall (n, n') \in M'_{i,j}, \tau_{i.n} \rightarrow \tau_{j.n'}$$

avec

$$M'_{i,j} = \{(n,n')|\exists k \in \mathbb{N}, (m,m') \in M_{i,j}, (n,n') = (m,m') + (k \cdot \frac{p \cdot L_{i,j}}{T_i}, k \cdot \frac{p \cdot L_{i,j}}{T_j})\}$$

Une précédence simple $\tau_i \to \tau_j$ est un cas particulier de contrainte de précédence périodique étendue où : $L_{i,j} = 1$, $M_{i,j} = \{(0,0)\}$ et τ_i et τ_j ont la même période. Les opérateurs de base $op ::= /\hat{}\ k \mid *\hat{}\ k \mid \sim> q \mid$ `tail` avec $q < 1$ appliqués seuls, i.e. $\tau_j(op(\tau_i(x)))$ génèrent les contraintes de précédence $M_{i,j} = \{(0,0)\}$ avec $L_{i,j} = 1$ (cf figure 2.4(a)). Les opérateurs `fby` et $\sim> q$ avec $q \geq 1$ ne génèrent aucune contrainte cf 2.4(b). L'opérateur `::` n'ajoute aucune contrainte supplémentaire. L'enchaînement des opérateurs *op* génère des motifs de précédence plus complexes, cf 2.4(c).



(a) $\tau_2(\tau_1(x))$ - $M_{i,j} = \{(0,0)\}$

(b) $\tau_2(0$ `fby` $\tau_1(x))$ - $M_{i,j} = \emptyset$

(c) $\tau_2(\tau_1(x) *\hat{}\ 2/\hat{}\ 3)$ - $M_{i,j} = \{(0,0),(1,1)\}$

FIGURE 2.4 – Contraintes de précédence dues aux opérateurs de base

Exemple 5. *Pourquoi est-il nécessaire d'introduire l'entier $L_{i,j}$? Considérons l'exemple suivant :*

```
node ex (i : rate(6,0)) returns (o)
var z ;
let
  z = tau_1(i);
  o = tau_2((z/^3)*^3);
tel
```

Bien que les deux tâches τ_1 et τ_2 soient à la même période, la relation de précédence n'est pas une précédence simple mais une précédence toutes les 3 exécutions : $\tau_{1.3k} \to \tau_{2.3k}$. Il faut donc regarder sur l'intervalle $3 \times ppcm(T_1, T_2)$ et utiliser $L_{1,2} = 3$. La contrainte de précédence est alors $\tau_1 \xrightarrow{\{0,0\},3} \tau_2$.

Exemple 6. *Le programme de l'exemple 1 p. 12, produit la précédence simple $\tau_1 \to \tau_3$ et la précédence étendue $\tau_2 \xrightarrow{\{0,0\},1} \tau_3$ avec $L_{2,3} = 1$. Dans le premier cas, il s'agit d'une précédence simple car les tâches τ_1 et τ_3 ont la même période. Tandis que τ_2 et τ_3 n'ayant pas la même période, $\tau_2 \xrightarrow{\{0,0\},1} \tau_3$ est bien une contrainte de précédence entre instances de tâches, qui produit $\tau_{2.6k} \to \tau_{3.k}$.*

Les précédences simples issues du programme `cdv` *sont $F_A \to L_A$, $F_P \to L_P$, et $F_G \to L_G$. Les autres sont $A_P \xrightarrow{\{(0,0),(2,1),(4,2)\},1} F_G$, $A_C \xrightarrow{\{(0,0),(1,1),(2,2)\},1} F_P$ et $L_P \xrightarrow{\{(0,0),(1,2),(2,3)\},1} L_A$.*

Propriété 1 (Lien entre mot de dépendance de données et précédences étendues)**.** *On peut retrouver le motif de précédence étendue à partir des mots de dépendance de données. Si précédence il y a pour une consommation, elle a forcément lieu pour le premier consommateur. Considérons l'expression $\tau_2(ops(\tau_1(i)))$, la consommation des données est exprimée par le mot $w = (-1,0)(j_1,d_1)\dots(j_n,d_n)$ ($d_0 = 0$ car si un* `fby` *est utilisé, on est assuré qu'il n'y aura pas de précédence). Le mot est donc exprimé sur l'intervalle $(\Sigma j_i) \times T_1 = (\Sigma d_i) \times T_2$. Ainsi*

$$L_{i,j} = \frac{(\Sigma j_i) \times T_1}{ppcm(T_1, T_2)}$$

La première instance consommatrice d'une donnée produite génère une précédence si le producteur n'a pas nécessairement terminé son exécution au réveil du consommateur, c'est-à-dire (NB ne pas oublier que les numéros des instances commencent à 0 et celles des données à 1) :

$$\begin{aligned}
&\forall k \in \mathcal{I}_{p \cdot L_{i,j}/T_i}, \forall k' \in \mathcal{I}_{p \cdot L_{i,j}/T_j} \\
&(k,k') \in M_{i,j} \iff \exists l \leq n, k = \Sigma_{i \leq l} j_i \wedge k' = \Sigma_{i \leq l-1} d_i + 1 \wedge \tau_{1.k-1} \to \tau_{2.k'-1} \\
&\qquad \iff \exists l \leq n, k = \Sigma_{i \leq l} j_i \wedge k' = \Sigma_{i \leq l-1} d_i + 1 \wedge (k-1) \times T_1 + O_1 + D_1 < (k'-1) \times T_2 + O_2
\end{aligned}$$

Exemple 7. *Considérons le cas d'une communication simple $\tau_2(\tau_1(i))$. Dans ce cas, le mot de dépendance de données est $(-1,0)(1,1)(1,1)$. Il exprime les communications sur $2 \times T_1 = 2 \times T_2$, ce qui donne $L_{1,2} = 2$. De plus, on a $\tau_{1.0} \to \tau_{2.0}$ et $\tau_{1.1} \to \tau_{2.1}$. On obtient alors la contrainte de précédence $M_{1,2} = \{(0,0),(1,1)\}$. C'est une version dépliée sur $2 \times ppcm$ de la contrainte de base montrée dans la figure 2.4(a).*

2.3.2 Protocole de communication

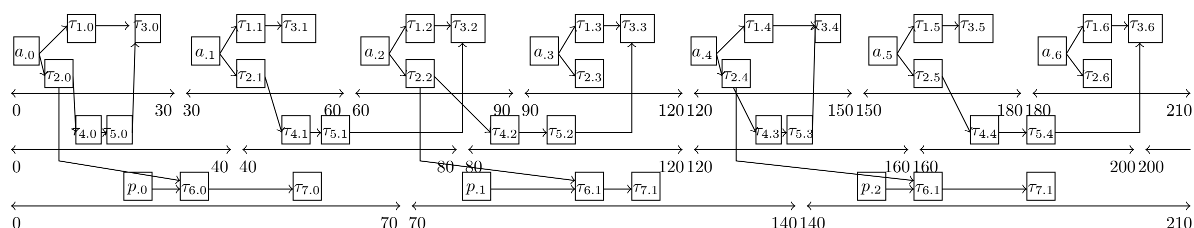
Afin que l'implantation multi-tâches d'un programme soit conforme à la sémantique du programme initial, il faut non seulement assurer le respect des précédences mais également utiliser un protocole de communication statique périodique permettant aux tâches de consommer les bonnes valeurs des flots.

Problème des entrelacements. Pour conserver la sémantique synchrone, il faut que les instances consomment les données produites par les bonnes instances productrices. Comme des instances communicantes ne s'exécutent pas forcément au même rythme, il se peut qu'il faille introduire des mémoires de communication supplémentaires.

Exemple 8. *Considérons l'exemple suivant :*

```
node assemblage (a : rate (30, 0);  p: rate (70, 0))
returns (o)
var p_i, a_2, a_3;
let
  p_i = tau_2(a);
  o   = tau_3(tau_1(a), (a_2*^4)/^3);
  a_2 = tau_5 (tau_4(p_i*^3/^4),((0 fby a_3)*^7)/^4);
  a_3 = tau_7(tau_6(p_i*^3/^7),p);
tel
```

Le comportement est décrit ci-dessous :



Les données produites par τ_2 sont consommées à la fois par τ_4 et τ_6. La première valeur est utilisée par $\tau_{4.0}$ et $\tau_{6.0}$, elle doit donc être disponible jusqu'à la fin de l'exécution de ces deux instances. Comme $\tau_{4.0}$ termine au plus tard à 40 et $\tau_{6.0}$ à 70, soit on considère que la première donnée doit être disponible pendant toute la période de τ_6 soit 70 unités de temps, soit on veut optimiser et décider que la valeur doit être disponible pendant le pire temps de réponse des tâches consommatrices. Nous faisons le choix de la solution 1, de même que Sofronis et al., car l'optimisation de la solution 2 nous semble coûteuse et non nécessairement efficace. La deuxième valeur est consommée par $\tau_{4.1}$ entre 40 *et* 80. *Il faut donc un buffer de taille 2 pour mémoriser les valeurs produites par τ_2 au profit de τ_4 et τ_6.*

La tâche τ_5 communique avec τ_3. La donnée produite par $\tau_{4.0}$ est consommée par $\tau_{3.0}$ et $\tau_{3.1}$. La deuxième instance de τ_5 peut s'exécuter avant $\tau_{3.1}$, il faut donc deux cases : une pour stocker la valeur précédente et une pour écrire la valeur courante. Enfin la tâche τ_7 communique avec τ_5 avec un délai pour lequel il faut également 2 cases. La figure 2.5 montre les buffers de communication.

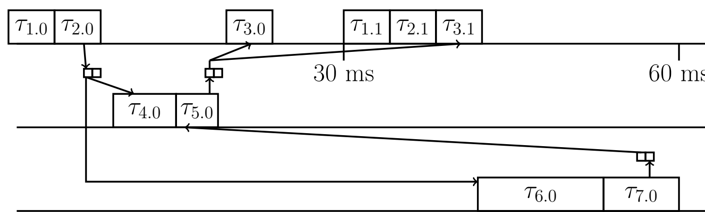


FIGURE 2.5 – Exemple de problème d'entrelacement

Protocole de communication. Les auteurs de [STC06 ; CSST08] ont défini le protocole *Dynamic Buffering Protocol* (DBP) pour assurer les communications d'un ensemble de tâches temps réel obtenues à partir d'un code LUSTRE. Toutes les implantations multi-tâches de langages synchrones, dont la nôtre, sont inspirées de ce protocole. DBP est applicable notamment à des tâches apériodiques et décrit les accès des tâches en lecture / écriture aux buffers par l'intermédiaire de pointeurs. Les pointeurs sont modifiés aux dates de réveil de certaines instances de tâche.

Le contexte multi-périodique étant plus restreint, les schémas d'accès aux buffers sont connus statiquement. Notre approche, détaillée dans [BCF09 ; FBLP10 ; PFB11], consiste à implanter une version statique et sans pointeur du protocole. Soit $\tau_1 \rightarrow^{ops} \tau_2$, le mot de dépendance associé à cette communication $w = (-1, d_0)(j_1, d_1) \dots (j_n, d_n)$ décrit les instances de τ_1 dont les valeurs sont consommées par des instances de τ_2. Lorsque les valeurs produites par une instance de τ_1 sont consommées : le premier consommateur génère la relation de précédence, comme illustré dans la propriété 1, tandis que le dernier consommateur permet de déterminer le dernier instant de consommation d'une donnée.

$$\forall h \text{ tel que } \exists l \leq n, h = \Sigma_{i \leq l} j_i, \text{ le dernier consommateur est } \tau_{1.h-1} \rightarrow \tau_{2.h'-1} \text{ avec } h' = \Sigma_{i \leq l} d_i$$

Ainsi, $\forall l \leq n$, soit $h = \Sigma_{i \leq l}\, j_i$ et $h' = \Sigma_{i \leq l}\, d_i$:

$$\begin{array}{lll} \tau_{1.h-1} & \textit{écrit dans l'intervalle} & [(h-1) \times T_1 + O_1, (h-1) \times T_1 + D_1 + O_1] \\ \tau_{2.h'-1} & \textit{consomme la donnée dans} & [(h'-1) \times T_2 + O_2, (h'-1) \times T_2 + D_2 + O_2] \end{array}$$

Alors

$$\tau_{1.h-1} \quad \textit{doit être disponible jusqu'à} \quad (h'-1) \times T_2 + D_2 + O_2$$

Le compilateur PRELUDE génère un buffer par donnée échangée et calcule la taille de ce buffer. Les contraintes sur les dates de disponibilité ci-dessus permettent de calculer le nombre de cases par buffer nécessaires dans une communication. Le compilateur produit également un code statique d'empaquetage des tâches décrivant les cases des buffers où chaque instance de tâche doit lire et écrire ses données. Cela se représente par des patrons répétitifs d'accès de données de la forme :

$$pat = \langle r_1, \dots, r_n \rangle, \ r_i \in \mathbb{N}$$

où $r_i = 0$ signifie pas d'accès au buffer et $r_i > 0$ accès à la case r_i.

Exemple 9. *Considérons à nouveau l'exemple 1 p. 12. Pour la précédence simple $\tau_1 \rightarrow \tau_3$, le mot de dépendance de données est $(-1,0)(1,1)(1,1)$. τ_1 et τ_3 ont pour période 24, mais τ_3 a un décalage de phase de 12 ce qui signifie qu'une exécution de τ_3 chevauche deux exécutions de τ_1. Il faut donc un buffer de taille 2. A chaque exécution, τ_1 écrit dans la case opposée à la précédente et τ_3 lit en suivant le même patron. Donc le patron en écriture pour τ_1 est $\langle 1, 2 \rangle$ et celui en lecture de τ_3 est $\langle 1, 2 \rangle$.*

Considérons la deuxième précédence $\tau_2 \xrightarrow{\{0,0\},1} \tau_3$. Le mot de dépendance de données est $(-1,0)(1,1)$ $(6,1)$, τ_2 est de période 4 et donc chaque exécution de τ_3 ne croise qu'une unique instance de τ_2 consommée. Il faut donc un buffer de taille 1 où $\tau_{2.6k}$ doit être disponible dans l'intervalle $[24 \times k, 24 \times (k+1) + 12]$. τ_2 n'écrit pas toutes les valeurs dans le buffer mais seulement toutes les 6 exécutions, donc le motif en écriture est $\langle 1, 0, 0, 0, 0, 0 \rangle$. τ_3 en revanche lit la valeur dans le buffer toutes les exécutions, donc le motif en lecture est $\langle 1 \rangle$.

Optimisation du nombre de buffers. Il est possible d'optimiser le nombre de buffers et de cases pour les communications. Sofronis et al. [CSST08] ont décrit un algorithme pour calculer le nombre optimal de buffers nécessaires pour des tâches multi-périodiques. L'idée est la suivante : pour l'ensemble des communications $\tau_1 \rightarrow \tau_2, \dots, \tau_1 \rightarrow \tau_n$, on regarde sur l'hyper-période des tâches et pour chaque instant d'écriture de τ_1 si :

1. la donnée produite est consommée,
2. si oui, on teste s'il est possible de réutiliser une case existante du buffer de communication pour stocker cette donnée. Si aucune case ne peut être réutilisée, on ajoute 1 au nombre de cases.

Cette idée peut facilement être adaptée au contexte des mots de dépendance de données comme montré dans [BCF09 ; PFB11]. Pour toutes les consommations de τ_1, $\tau_1 \rightarrow^{ops} \tau_k$, $k = 2, \dots, m$, on obtient m mots de dépendance $w^k = (-1, d_0^k) \dots (j_n^k, d_{n_k}^k)$. Il faut raisonner sur l'hyper-période de l'ensemble de ces

mots, à savoir $H = ppcm_k((\Sigma_l j_l^k) \cdot T_1)$. On déplie alors les mots périodiques pour qu'ils soient tous de même taille (ce *ppcm*). On calcule ensuite un tableau représentant le protocole de communication entre τ_1 et ses consommateurs de longueur *ppcm* dont chaque élément est de la forme : *(numéro de l'instance de τ_1 consommée, numéro du buffer, liste des consommateurs et numéro des instances, date absolue de fin de consommation)*. La première instance consommée $j_1 = min_k j_1^k$ et on obtient le tableau $[j_1$, 1, liste de consommateurs, $max_{k|j_1^k=j_1}((d_0^k + d_1^k - 1) \times T_k + D_k + O_k)]$. Pour chaque nouvelle instance consommée j_k, on regarde si l'un des buffers p est disponible dans le tableau, si oui on rajoute dans le tableau $[j_k$, p, liste de consommateurs, plus grande date de consommation] sinon on ajoute un nouveau buffer $[j_k$, nb $+1$, liste de consommateurs, plus grande date de consommation].

Exemple 10. *Sur l'exemple des commandes de vol, on obtient 6 buffers au lieu de 38. Il faut en particulier 2 cases pour les communications $AP \rightarrow^{*\hat{}3,/\hat{}4} FP$ et $AP \rightarrow^{*\hat{}3,/\hat{}7} FG$. Les mots sont respectivement* $(-1,0)(1,1)(1,1)(1,1)(2,1)$ *et* $(-1,0)(1,1)(2,1)(2,1)\ (3,1)$. *Le premier motif porte sur les instances 2 à 5, et le deuxième les instances 3 à 8. Le motif d'écriture de AP est basé sur les* 40 *premières instances. La première instance de AP écrit dans la première case du buffer et sera consommée par les premières instances de FP et FG. Cela donne à l'initialisation du tableau [(1, 1, [(FP,1) ;(FG,1)], 70)] car* $D(FG) = 70$. *La deuxième instance est écrite dans la deuxième case car elle débute à 30 et est consommée par la deuxième instance de FP. Le calcul donne alors [(1, 1, [(FP,1) ;(FG,1)], 70) ; (2, 2, [(FP,2)], 80)]. La troisième instance est consommée par les deux tâches ce qui donne [(1, 1, [(FP,1) ;(FG,1)], 70) ; (2, 2, [(FP,2)], 80) ; (3, 1, [(FP,3) ;(FG,2)], 140)] et ainsi de suite jusqu'à 40.*

2.3.3 Modèle de tâches générées

Le compilateur PRELUDE produit donc un ensemble de tâches périodiques reliées par des contraintes de précédence étendues et communicantes de la forme $\mathcal{P} = \langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$ avec :

- $\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$ est un ensemble fini de tâches (correspondant aux nœuds importés). Chaque tâche est définie par $\tau_i = (O_i, T_i, C_i, D_i)$;
- $\mathcal{R}$ est une relation de précédence étendue liant les jobs entre eux, i.e.

$$\mathcal{R} = \{\tau_i \xrightarrow{M_{i,j}, L_{i,j}} \tau_j\}$$

- $\mathcal{C} = \{b_1, \ldots, b_k\}$ est un ensemble fini de buffers échangés entre les tâches τ_i. Pour chaque buffer, on connaît le producteur, le consommateur, le nombre de cases, les patrons en lecture et écriture.

2.3.4 Implantation mono-processeur sans sémaphore

J. Forget a réalisé dans sa thèse un premier portage de programmes PRELUDE sur des architectures mono-processeurs en utilisant MARTE OS [RH02], un ordonnanceur EDF pré-emptif et un encodage des précédences en réutilisant les résultats de Chetto et al. [CSB90]. Les buffers sont implantés par des variables partagées et sont en dehors de portée des questions d'ordonnancements. La question est donc d'implanter un ensemble de tâches $\mathcal{P} = \langle \mathcal{S}, \mathcal{R} \rangle$.

Encodage des précédences

Chetto et al. [CSB90] ont proposé un algorithme pour transformer un ensemble de tâches apériodiques avec précédence en un ensemble de tâches indépendantes en modifiant les dates de réveil et les échéances ; et ont montré que les deux ensembles de tâches sont équivalents pour EDF, i.e. les deux ensembles sont ordonnançables ou aucun des deux ne l'est. Pour toute tâche τ, la modification des dates de réveil se fait de la façon suivante : $O^\star = max_{j \mid \tau_j \rightarrow \tau}(O, O_j^\star + C_j)$. A cause de la variabilité des temps d'exécution, on peut légèrement modifier cette règle par $O^\star = max_{j \mid \tau_j \rightarrow \tau}(O, O_j^\star)$ pour ne pas introduire de temps creux. L'algorithme agit ensuite sur l'échéance absolue de toute tâche τ : $d^\star = min_{j \mid \tau \rightarrow \tau_j}(d, d_j^\star - C_j)$. Ces formules peuvent aisément être transposé dans le modèle de tâches de PRELUDE :

1. sur des échéances relatives. Il suffit d'adapter les formules. Par exemple, si $\tau_1 \rightarrow \tau_2$ alors $O_2^\star = max(O_2, O_1^\star)$ et $D_1^\star + O_1 + C_2 \leq D_2^\star + O_2^\star$, i.e. $D_1^\star = min(D_1, D_2^\star + O_2^\star - (O_1 + C_2))$.
2. sur des contraintes de précédence étendue. Les modifications doivent donc porter sur les dates de réveil et les échéances des instances des tâches. Nous avons donc introduit des mots de date de réveil et des mots d'échéances. Puisque les contraintes portent sur des patrons finis, on est assuré que ces mots sont finis. L'idée est la suivante : soit $\tau_1 \rightarrow^{ops} \tau_2$, on calcule alors le mot de

dépendance associé à cette communication $w = (-1, d_0) \dots (j_n, d_n)$. On a en particulier $\forall l \leq n$, $h = \Sigma_{i \leq l} j_i$, $h' = \Sigma_{i \leq l-1} d_i + 1$, $\tau_{1.h-1} \to \tau_{2.h'-1}$. On note $O(\tau_{i.j}) = O_i$ et on a $o_{i.j} = O_i + j \cdot T_i$. On obtient alors :

$$\begin{cases} O^\star(\tau_{2.h'-1}) = max(O_2, O^\star(\tau_{1.h-1}) + (h-1) \cdot T_1 - (h'-1) \cdot T_2) \\ D^\star(\tau_{1.h-1}) = min(D_1, D_2^\star(\tau_{2.h'-1}) + O^\star(\tau_{2.h'-1}) - (O^\star(\tau_{1.h-1}) + C_2)) \end{cases}$$

Grâce à cet encodage, tout programme $\mathcal{P} = \langle \mathcal{S}, \mathcal{R} \rangle$ peut être transformé en un ensemble de tâches indépendantes $\mathcal{P}^\star = \langle \mathcal{S}^\star \rangle$ équivalent.

Généralisation à d'autres politiques d'ordonnancement

Dans [FBG10], nous avons réutilisé l'encodage mentionné ci-dessus et montré qu'il peut être réutilisé pour des politiques à priorité fixe. Nous avons considéré trois cas de figure :

1. ensemble de tâches synchrones avec précédences simples : en modifiant légèrement l'encodage de Chetto et al. [CSB90], le système obtenu est équivalent au système initial dans le sens où

   $\mathcal{S}$ est faisable si et seulement si $\mathcal{S}^*$ est faisable

   De ce fait, la politique DM (deadline monotonic) est optimale pour les ensembles de tâches considérés.
2. ensemble de tâches asynchrones avec précédences simples : nous avons à nouveau montrer l'équivalence après application de l'encodage. Nous avons ensuite étendu le test de faisabilité d'Audsley [Aud91] afin de considérer les précédences.
3. ensemble de tâches asynchrones avec précédences étendues : soit $\mathcal{S} = \{\tau_i = (O_i, T_i, D_i, C_i)\}$ avec les contraintes de précédence étendue $M = \{M_{i_1,j_1}, \dots, M_{i_l,j_l}\}$. Le système après encodage (le même que présenté ci-dessus) $\mathcal{S}^* = \{\tau_i'(O_i^*, T_i, D_i^*, C_i)\}$ avec les précédences $\to = \{(\tau_{i_k}, \tau_{j_k}) | k = 1, \dots, l\}$ est équivalent au système initial. On peut donc ensuite appliquer l'algorithme d'Audsley modifié du point 2.

2.4 Utilisation de Prelude sur le cas d'étude Rosace

Nous avons collaboré avec David Saussié de Polytechnique Montréal afin d'utiliser Prelude sur un développement complet [PSG14], c'est-à-dire en suivant un processus de développement d'un contrôleur longitudinal de vol de la spécification Simulink jusqu'à l'exécution sur un pluri-cœurs. L'étude de cas s'appelle Rosace pour Research Open-Source Avionics and Control Engineering et est mise à la disposition de la communauté dans le dépôt svn `https://svn.onera.fr/schedmcore/branches/schedmcore-RTAS2014/Case_Study_RTAS`.

Objectifs

La phase de conception détaillée d'un contrôleur de vol pour de l'avionique civile certifiée peut très schématiquement être décrite par la figure 2.6. Les ingénieurs en automatique développent leurs modèles en Matlab/Simulink, analysent leur comportement et synthétisent des contrôleurs vérifiant des exigences de performances et de robustesse. Les parties spécifiques à notre approche sont écrites en italique. Nous avons identifié les exigences de performance qui concernent l'implantation sur la cible en liaison avec les choix temps réel (périodes, contraintes de précédence, délais sur le bus).

Du fait du besoin de certification et pour respecter le standard DO-178B [RTC08], le générateur de code de The Mathworks ne peut être utilisé car il n'est pas qualifié. Le contrôleur est alors re-codé en Scade (dont le générateur de code est qualifié), soit en utilisant un traducteur d'un sous-ensemble de Simulink vers Scade, soit intégralement à la main. Nous avons traduit à la main les blocs de base de l'étude de cas en Lustre puis nous avons utilisé le compilateur *lustrec* [GHKT14] développé par Pierre-Loïc Garoche et Xavier Thirioux pour obtenir les nœuds importés en C. L'assemblage multi-périodique est écrit en Prelude. Nous avons proposé plusieurs assemblages allant du plus contraint d'un point de vue temps réel (et équivalent à la spécification Simulink) jusqu'à des assemblages nettement moins contraints. Nous avons alors simulé les différents programmes Prelude et comparer leur comportement vis-à-vis des exigences de performances temps réel.

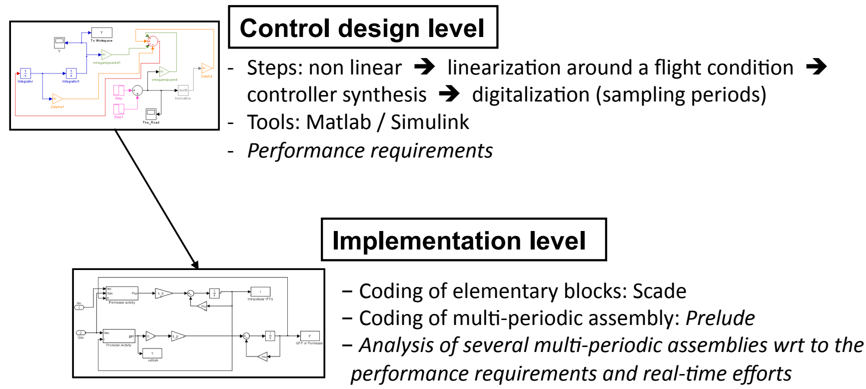


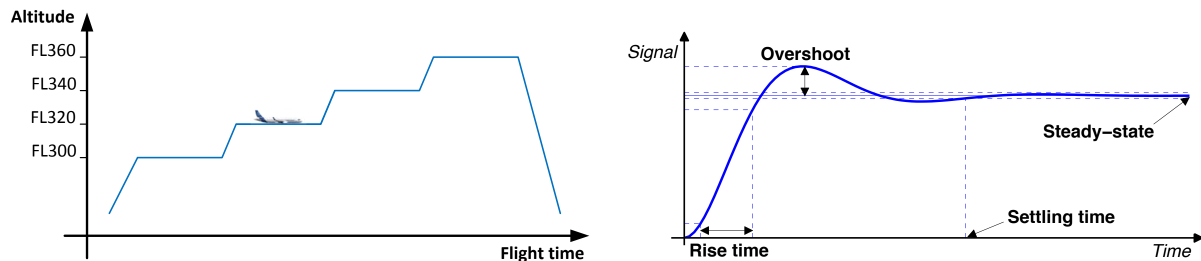
FIGURE 2.6 – Phase de conception détaillée d'un contrôleur de vol

Contrôleur de vol longitudinal

Le contrôleur longitudinal de vol est défini pour la phase *en-route* d'un avion civil de type moyen-courrier au point de vol $h = 10000\,\text{m}$, $V_a = 230\,\text{m/s}$. Cette étude de cas est une extension multi-périodique du contrôleur longitudinal mono-périodique de Gervais et al. [GCS12]. Un tel contrôleur a pour mission :

— de maintenir l'altitude h et la vitesse aérodynamique V_a constantes en phase de *croisière*,

— de changer de *niveau de vol* à la demande du pilote en appliquant une vitesse verticale V_z constante. Un niveau de vol est noté FLxxx (ex. FL300 représente l'altitude pression 30000 ft). Un exemple de changement de niveau de vol est donné dans la figure 2.7(a).



(a) Changement de niveaux de vol. Ici FL300 → FL320 → FL340 → FL360

(b) Exigences du contrôleur

FIGURE 2.7 – Description générale du contrôleur de vol ROSACE

Quatre exigences ont été identifiées (cf figure 2.7(b)) et devront être assurées par l'exécution réelle. Les valeurs objectif associées à ces exigences sont exprimées dans une approche dite découplée, c'est-à-dire que l'on ne donne une consigne que sur V_z ou V_a. Les objectifs sont décrits dans la table 2.1.

Exigence	Description	Objectifs	
P1 *Temps de réponse (Settling time)*	temps nécessaire pour rester à + ou - 5% de la consigne	V_z	$\leq 10\,\text{s}$
		V_a	$\leq 20\,\text{s}$
P2 *Dépassement (Overshoot)*	valeur maximale atteinte moins la valeur en régime permanent	V_z	$\leq 10\%$
		V_a	$\leq 10\%$
P3 *Temps de montée (Rise time)*	temps nécessaire pour passer de 10% à 90% de la valeur en régime permanent	V_z	$\leq 6\,\text{s}$
		V_a	$\leq 12\,\text{s}$
P4 *Erreur en régime permanent (Steady-state error)*	différence entre la consigne et la valeur en régime permanent quand $t \to \infty$	V_z	$\leq 5\%$
		V_a	$\leq 5\%$

TABLE 2.1 – Exigences de performance

Le schéma bloc SIMULINK, décrit dans la figure 2.8, est la version linéarisée discrétisée du modèle original. Il est composé de deux parties :

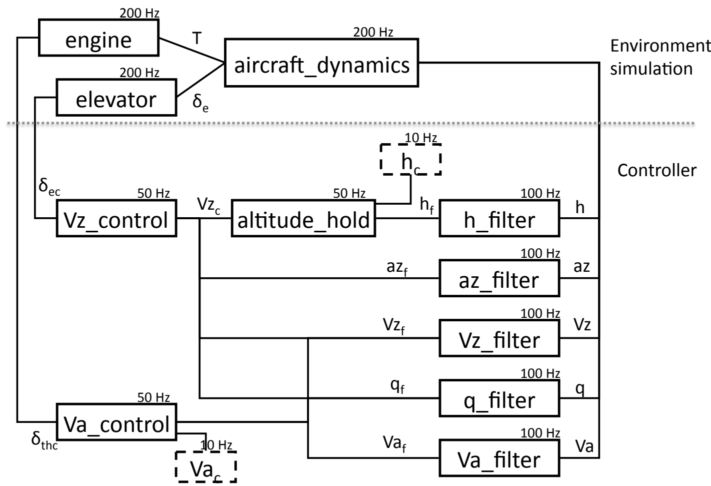


FIGURE 2.8 – Contrôleur de vol ROSACE

1. en haut, la partie *Environment Simulation* représente le modèle avion qui doit être contrôlé, c'est-à-dire l'avion (*aircraft_ dynamics*) ainsi que les moteurs (*engine*) et les gouvernes de profondeur (*elevator*) ;
2. le *Controller* proprement dit regroupe deux boucles de contrôle et des filtres dont le but est de consolider les données transmises par les capteurs. L'objectif du contrôleur longitudinal de vol est de suivre avec précision les consignes du pilote sur l'*altitude* (h_c), la poussée verticale (V_{z_c}) et la vitesse aérodynamique (V_{a_c}).
   — la première boucle de contrôle est composée d'*altitude_ hold* et *Vz_ control*. Le contrôle de l'altitude est découpé en deux phases : la commande en altitude h_c est d'abord traduite en une commande de vitesse verticale V_{z_c} par *altitude_ hold* et *Vz_ control* est en charge de suivre cette consigne. Durant une phase de montée, la logique du contrôleur est la suivante : une vitesse verticale constante de $V_{z_c} = 2.5\,\text{m/s}$ est d'abord imposée à l'avion jusqu'à atteindre 50 m sous le niveau de vol demandé. A cette distance, *altitude_ hold* prend la main pour commander plus finement la vitesse verticale (et donc l'altitude) et ne pas susciter d'inconfort pour les passagers.
   — la deuxième boucle *Va_ control* maintient V_a constant ou suit la consigne V_{a_c}.

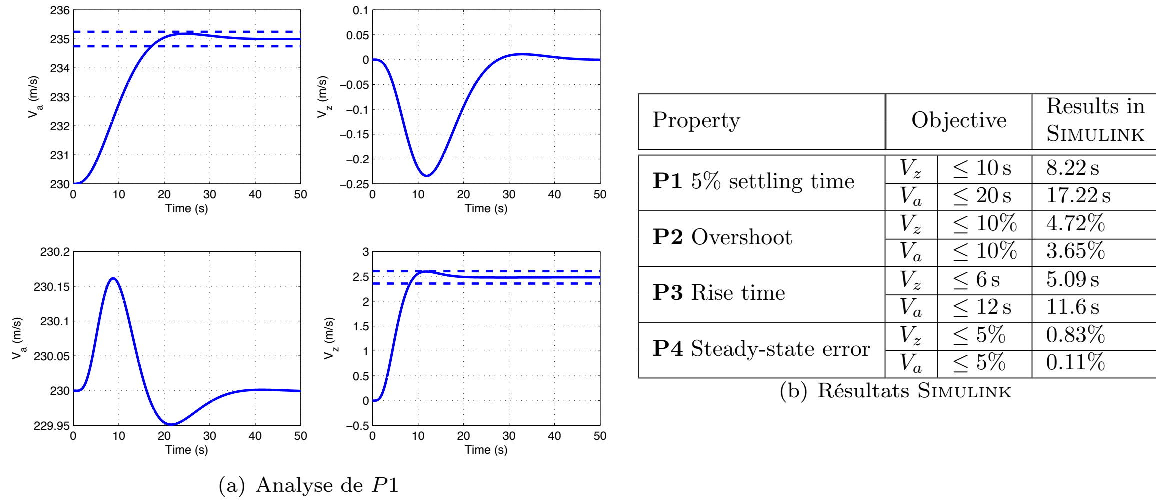
Les fréquences mentionnées sur le schéma SIMULINK ont été choisies par David Saussié :
— pour les contrôleurs : le modèle continu peut supporter un délai pur d'1 s avant déstabilisation donc la période d'échantillonnage doit être plus rapide que 1Hz. Les exigences de performance entraînent que la période d'échantillonnage doit être plus rapide que 10Hz ;
— pour les filtres : ces derniers doivent consommer suffisamment de données produites par les capteurs ;
— pour l'environnement : il faut être suffisamment rapide pour représenter un phénomène continu et être plus rapide que toutes les fréquences du contrôleur ce qui a amené à un choix de 200Hz.

La validation des quatre exigences en SIMULINK se fait par simulation. La figure 2.9(a) montre l'analyse découplée sur V_a et V_z. En haut à gauche, un ordre de 5m/s a été donné en V_{a_c}. On peut alors mesurer le *settling time* et observer en même temps l'effet en V_z. Ce dernier doit être minimal. De la même manière, en bas, on donne une consigne en V_{z_c} de 2.5m/s La figure 2.9(b) résume les valeurs obtenues par simulation et montre que les objectifs sont respectés. On peut donc passer à la phase de conception détaillée.

Implantation

Le codage des blocs élémentaires en LUSTRE a été fait à la main. A noter que la discrétisation de plusieurs blocs, notamment les filtres, dépend de la période d'échantillonnage. De ce fait, pour une autre période, il faut re-générer les coefficients du nœud.

L'assemblage a été écrit en PRELUDE. On choisit l'horloge de base avec une période de $100\mu s$. Ce choix est laissé à l'intégrateur et il doit prendre en compte plusieurs éléments :



(a) Analyse de $P1$

Property	Objective		Results in SIMULINK
P1 5% settling time	V_z	$\leq 10\,\mathrm{s}$	8.22 s
	V_a	$\leq 20\,\mathrm{s}$	17.22 s
P2 Overshoot	V_z	$\leq 10\%$	4.72%
	V_a	$\leq 10\%$	3.65%
P3 Rise time	V_z	$\leq 6\,\mathrm{s}$	5.09 s
	V_a	$\leq 12\,\mathrm{s}$	11.6 s
P4 Steady-state error	V_z	$\leq 5\%$	0.83%
	V_a	$\leq 5\%$	0.11%

(b) Résultats SIMULINK

FIGURE 2.9 – Validation en SIMULINK

1. point de vue applicatif : toutes les périodes des tâches et les pires temps d'exécution sur la cible doivent être multiples de l'horloge de base ;
2. point de vue matériel : la période base doit être mesurable par les horloges fournies par le matériel.

Les entrées de l'assemblage sont les consignes pilote, h_c and V_{a_c}, ainsi que les sorties du modèle avion, V_a, V_z, q, a_z, h. Nous avons choisi d'implanter le système complet pour valider l'exécution, ainsi les sorties de l'avion sont considérées comme des variables intermédiaires. Il faut spécifier en PRELUDE le rythme des entrées, dans notre cas, on suppose que ces entrées sont à 10Hz et donc en horloge strictement périodique cela donne $(1000, 0)$ pour une horloge de base à $100\mu s$. Les sorties du nœud sont les ordres envoyés aux gouvernes, c'est-à-dire δ_{e_c} et δ_{th_c}. L'assemblage est décrit dans la figure 2.10.

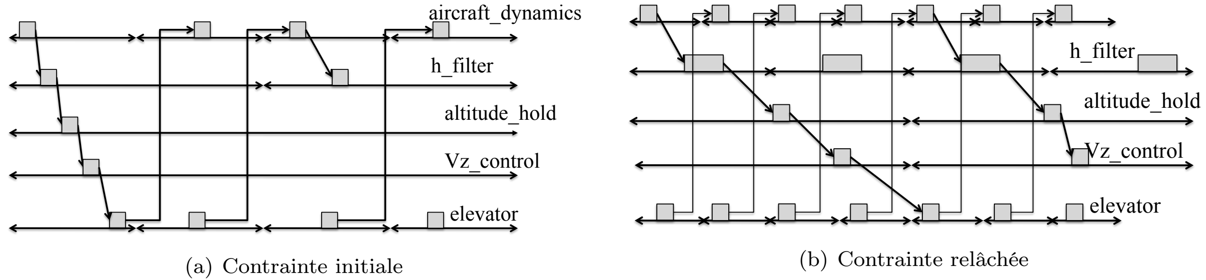
```
node assemblage (h_c, Va_c: real rate (1000,0))
returns (delta_e_c, delta_th_c : real)
var Va, Vz, q, az, h : real;
   Va_f, Vz_f, q_f, az_f, h_f,  Vz_c, delta_e, T : real;
let
  h_f = h_filter(h/^2);
  Va_f = Va_filter(Va/^2);
  Vz_f = Vz_filter(Vz/^2);
  q_f = q_filter(q/^2);
  az_f = az_filter(az/^2);
  Vz_c = altitude_hold (h_f/^2, h_c*^5);
  delta_th_c = Va_control (Va_f/^2, Vz_f/^2, q_f/^2, Va_c*^5);
  delta_e_c = Vz_control (Vz_f/^2, Vz_c, q_f/^2, az_f/^2);
  T = engine(delta_th_c*^4);
  delta_e = elevator(delta_e_c*^4);
  (Va, Vz, q, az, h) =
        aircraft_dynamics((0.018645918123716 fby delta_e), (43219.8575 fby T));
tel
```

FIGURE 2.10 – Codage en PRELUDE de l'assemblage

Le calcul d'horloge appliqué à ce nœud donne les résultats suivants (en accord avec la figure 2.8) :

Nœud	Fréquence	Horloge	Nœud	Fréquence	Horloge
altitude_hold	50Hz	$(200, 0)$	*h_filter*	100Hz	$(100, 0)$
Va_filter	100Hz	$(100, 0)$	*q_filter*	100Hz	$(100, 0)$
az_filter	100Hz	$(100, 0)$	*Va_control*	50Hz	$(200, 0)$
Vz_control	50Hz	$(200, 0)$	*engine*	200Hz	$(50, 0)$
elevator	200Hz	$(50, 0)$	*ac_dynamics*	200Hz	$(50, 0)$

Cette première implantation est très contraignante d'un point de vue temps réel. En effet, on a réduit les fréquences afin de restreindre l'occupation CPU mais les précédences imposées par l'assemblage présentent des chaînes fonctionnelles qui doivent s'exécuter dans des intervalles très petits. Par exemple, la chaîne fonctionnelle $h \rightarrow h_f \rightarrow V_{z_c} \rightarrow \delta_{e_c} \rightarrow \delta_e$ produit l'exécution montrée dans la figure 2.11(a). La séquence de tâches *aircraft_ dynamics*, *h_filter*, *altitude_ hold*, *Vz_ control*, *elevator* doit se faire en moins de 5ms toutes les 20 ms.



(a) Contrainte initiale

(b) Contrainte relâchée

FIGURE 2.11 – Deux implantations de la chaîne fonctionnelle $h \rightarrow h_f \rightarrow V_{z_c} \rightarrow \delta_{e_c} \rightarrow \delta_e$

Afin de réduire les contraintes d'intégration, nous avons programmé un deuxième assemblage dont les contraintes de précédence sont montrées figure 2.11(b). Grâce à cette nouvelle spécification, le système est davantage parallélisable. Pour spécifier ce nouvel assemblage, il suffit de changer les deux équations suivantes :

```
T = engine ((1.640222296162316 fby delta_th_c)*^4);
delta_e = elevator ((0.018645918123716 fby delta_e_c)*^4);
```

Réaliser le même comportement en SIMULINK est assez complexe.

Nous avons également testé un autre assemblage qui réduit les fréquences : *h_filter*, *Va_filter*, *Vz_filter* passent à 50Hz et *altitude_ hold* passe à 10Hz. Pour les trois filtres, nous devons changer le rythme sur les flots d'entrée comme suit :

```
h_f = h_filter(h/^4);
```

Pour *altitude_ hold*, l'équation devient

```
Vz_c = altitude_hold (h_f/^5, h_c);
```

L'équation de *Vz_ control* doit aussi être modifiée comme suit :

```
delta_e_c = Vz_control(Vz_f, Vz_c*^5, q_f/^2, az_f/^2);
```

On a alors appliqué les mêmes méthodes de validation qu'au niveau SIMULINK, c'est-à-dire simulation sur plusieurs secondes avec plusieurs jeux d'entrée. Le simulateur PRELUDE est fourni par SCHEDMCORE décrit dans le chapitre 3. Nous avons implanté un script qui parcourt les simulations produites pour vérifier les propriétés. Pour les trois assemblages considérés, les comportements sont corrects vis-à-vis des propriétés de performance.

2.5 Vers une spécification simplifiée

L'étude de cas ROSACE illustre parfaitement les avantages et les inconvénients du langage. PRELUDE est bien adapté à la spécification formelle de programmes multi-périodiques et permet d'exprimer des combinaisons complexes de manière simple. La suite d'outil SCHEDMCORE fournit un moyen de simulation afin de valider et observer les comportements fonctionnels. Le compilateur génère ensuite un ensemble de tâches dépendantes assez standard qu'il est aisé d'ordonnancer par n'importe quel ordonnanceur connu. Néanmoins, les choix au niveau assemblage sont parfois difficiles. L'objectif de la thèse de Rémy Wyss est d'aider le concepteur dans le choix des assemblages. En effet,

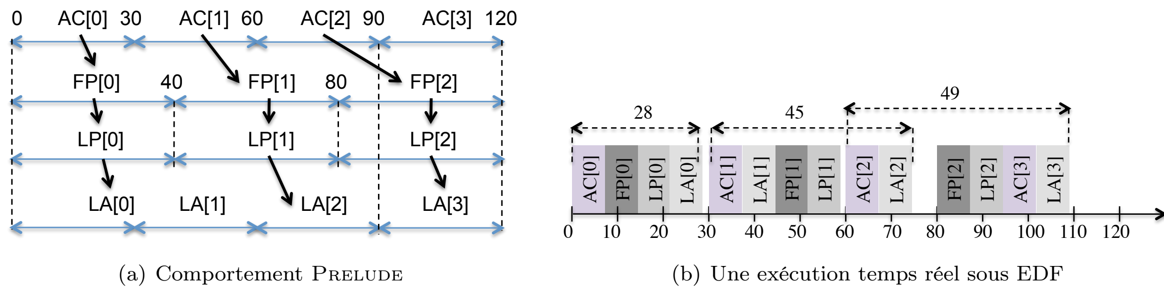
1. il n'est pas toujours simple de décider s'il faut mettre un `fby` , un décalage de phase ~> ou une communication directe ;
2. il est assez fastidieux d'écrire des assemblages de 5000 nœuds.

Son travail consiste donc à trouver des spécifications de plus haut niveau et non nécessairement complètes qui se traduisent ensuite en des programmes PRELUDE respectant les contraintes utilisateurs. L'approche proposée est de montrer comment calculer une latence maximale sur une chaîne fonctionnelle en PRELUDE et d'étendre le langage pour permettre d'exprimer des spécifications incomplètes avec un opérateur particulier appelé « don't care » qui sera instancié par une suite d'opérateurs PRELUDE respectant les exigences sur des latences de chaînes fonctionnelles. Ce travail est toujours en cours.

2.5.1 Latence de chaînes fonctionnelles

Motivation

Reprenons l'exemple initial des commandes de vol simplifiées figure 1.1 dont un assemblage a été proposé p 13. Cet assemblage génère notamment les précédences illustrées dans la figure 2.12(a) : la première valeur produite par AC est consommée par la première instance de *FP*, qui produit en chaîne une valeur pour la première instance de *LP*, qui à son tour transmet une valeur à la première instance de *LA*, ce qui se résume globalement par les précédences $AC_{.0} \rightarrow FP_{.0} \rightarrow LP_{.0} \rightarrow LA_{.0}$. Le schéma $AC_{.j} \rightarrow FP_{.j}$ se reproduit aux itérations suivantes, sauf pour $AC_{.3}$ dont la donnée n'est pas consommée. Comme $LA_{.1}$ ne consomme pas de nouvelles données venant de LP, la précédence suivante est $LP_{.1} \rightarrow LA_{.2}$. Le comportement est ensuite répétitif au delà de l'hyper-période. La figure 2.12(b) montre une exécution correcte sur un processeur avec la politique d'ordonnancement EDF.



(a) Comportement PRELUDE

(b) Une exécution temps réel sous EDF

FIGURE 2.12 – Comportement de la boucle d'asservissement

Pour exprimer l'enchaînement de plusieurs nœuds, nous avons déjà mentionné la notion de *chaîne fonctionnelle*, figure 2.11(a). Dans l'exemple ci-dessus, $AC \rightarrow FP \rightarrow LP \rightarrow LA$ est une telle chaîne. La première question qui se pose est : « *peut-on calculer la latence maximale d'une chaîne fonctionnelle au niveau* PRELUDE *en faisant abstraction de l'ordonnancement choisi ?* » D'ordinaire, l'estimation des latences se fait au niveau le plus bas, après l'implantation du système sous une forme multi-tâches sur un système d'exploitation temps réel, et donc tard dans le cycle de développement. Si, à ce niveau les latences calculées ou mesurées se révèlent être trop grandes, la seule latitude restante est souvent d'essayer d'optimiser l'implantation du système, voire d'opter pour des composants (processeur, mémoire) plus rapides. Cette solution tardive est souvent limitée, en particulier dans le contexte des systèmes à ressources contraintes (avionique, automobile, etc.). De ce fait, il est souhaitable de pouvoir étudier la dynamique, et en particulier, la ou les latences du système, dès les phases de conception amont, avant l'implantation temps réel.

Il est possible de calculer des latences maximales au niveau d'une description PRELUDE et ce sans présupposer d'implantation temps réel particulière. Dans l'exemple de la figure 2.12, la latence de la chaîne fonctionnelle $AC \rightarrow FP \rightarrow LP \rightarrow LA$ est bornée par 60ms. Il s'agit d'une borne supérieure car la latence maximale pour EDF montrée dans la figure 2.12(b) est de 49 ms.

Latence d'une chaîne fonctionnelle

La méthode pour calculer la latence d'une chaîne fonctionnelle est publiée dans [WBPF13]. On note toute dépendance syntaxique par $x' \leftarrow x$ (se lit x' dépend de x) de la forme $x' \overset{ops}{\longleftarrow} x$ où $ops = op_1.op_2.\cdots.op_n$ représente la composition $op_n(\ldots(op_2(op_1(i))))$ avec $op \in \{0, \texttt{fby}, \hat{*}, /\hat{}, \sim>, \texttt{tail}, ::\}$ et 0 représente une dépendance immédiate (par ex. l'appel d'un nœud importé).

Définition 4 (Chaîne fonctionnelle)**.** *Une chaîne fonctionnelle* $(x_1, \ldots, x_n)$ *est une liste de flots d'un programme* PRELUDE *tels que* $\forall 1 \leq i < n$, $x_{i+1} \leftarrow x_i$.

Exemple 11. *Dans l'exemple des commandes de vol simplifiées, la chaîne décrite dans la figure 2.12(a) est* L=(acc, acc_i, acc_o, angle_c, ordre)*. Dans la figure 2.11, la chaîne fonctionnelle est* (h, h_f, V_{z_c}, δ_{e_c}, δ_e)*.*

Définition 5 (Latence d'une chaîne fonctionnelle)**.** *Soit la chaîne fonctionnelle $L = (I, x_1, \ldots, x_n, O)$, la latence de L est le temps de propagation nécessaire pour produire une sortie consistante O pour une entrée donnée I, c'est-à-dire le temps maximal pour consommer une première fois toute valeur de I.*

Dans l'exemple des commandes de vol, la chaîne L_1 =*(angle, angle_o, ordre)* est mono-périodique sur l'horloge $(30, 0)$. On a donc $\forall i \in \mathbb{N}^*$, $ordre[i] \leftarrow angle[i]$. D'un point de vue synchrone, la latence est « nulle ». D'un point de vue temps réel, la latence est dans l'intervalle $[0, 30]$ et on peut construire un ordonnancement tel que la latence soit 30.

Proposition 2. *Soit $L = (I, x_1, \ldots, x_m, O)$ une chaîne fonctionnelle. Par transitivité, nous avons $O \leftarrow^* I$; on note $w = (-1, d_0)\ (j_1, d_1) \ldots (j_n, d_n)$ le mot de dépendance de données représentant cette dépendance. La latence pire cas de L, notée $WCL(L)$, est :*

$$WCL(L) = \phi(O) - \phi(I) + \pi(O) + \pi(I) + \max_{k=1\ldots n} l_k$$

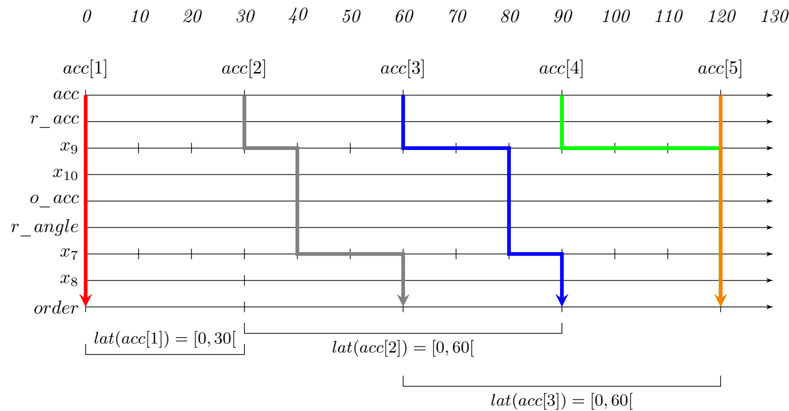
avec

$$\text{pour tout } k = 1 \ldots n, \quad l_k = \pi(O) \cdot \sum_{l=0}^{k-1} d_l - \pi(I) \cdot \sum_{l=1}^{k} j_l$$

Exemple 12. *Considérons la chaîne fonctionnelle L_2 = (acc, i_acc, x_9, x_{10}, o_acc, r_angle, x_7, x_8, $ordre$). On commence par calculer le mot de dépendance de données associé à la chaîne.*

$acc \rightarrow i_acc$	$(-1,0)(1,1)(1,1)$
$acc \xrightarrow{*} x_9$	$(-1,0)(1,3)(1,3)$
$acc \xrightarrow{*} x_{10}$	$(-1,0)(1,1)(1,1)(1,1)(2,1)$
$acc \xrightarrow{*} o_acc$	$(-1,0)(1,1)(1,1)(1,1)(2,1)$
$acc \xrightarrow{*} r_angle$	$(-1,0)(1,1)(1,1)(1,1)(2,1)$
$acc \xrightarrow{*} x_7$	$(-1,0)(1,4)(1,4)(1,4)(2,4)$
$acc \xrightarrow{*} x_8$	$(-1,0)(1,2)(1,1)(1,1)(2,2)$
$acc \xrightarrow{*} ordre$	$(-1,0)(1,2)(1,1)(1,1)(2,2)$

La figure ci-dessous montre les dépendances de données et illustre que la latence maximale obtenue en appliquant la formule de la proposition 2 est 60*.*



2.5.2 Assemblages mono-périodiques avec l'opérateur « don't care »

Motivation

L'extension pour permettre l'expression de spécifications incomplètes n'a été réalisée que sur des systèmes mono-périodiques pour lesquels l'instanciation de l'opérateur « don't care » est de la forme

`fby` ou communication directe. Ce travail a été publié dans [WBFP12a]. Le cas multi-périodique est plus complexe et ne sera pas discuté dans ce document. Considérons un nouvel exemple de commande de vol mono-périodique et comprenant des boucles figure 2.13. On reconnaît la structure des commandes de vol simplifiées figure 1.1 page 2 : les gouvernes ont été séparées en deux, où le contrôle des ailerons se trouve dans la partie droite de la figure et le contrôle des gouvernes de profondeur dans la partie gauche. Chaque boîte décrit un nœud importé et les communications représentées par les flèches sont de trois types : une flèche pleine noire représente une communication directe, une flèche pleine grise représente une communication avec un `fby` (il n'y en a pas dans la figure), et une flèche pointillée représente une communication « don't care ». Le guidage prend en entrée les consignes du pilote (*pos_ c*).

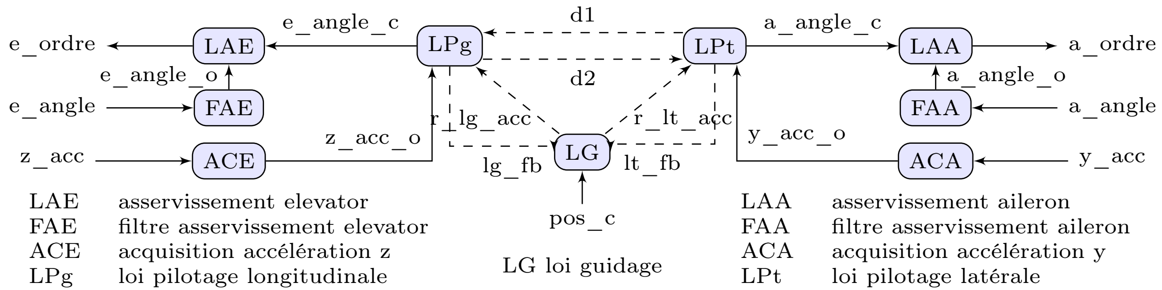


FIGURE 2.13 – Exemple de spécification incomplète

La variable *a_ angle* représente l'angle de l'aileron et est acquise par la fonction FAA (filtre asservissement aileron). Cette fonction consolide les données et envoie le flot *a_ angle_ o* (angle observé) à la fonction LAA (loi asservissement aileron). LAA contrôle l'aileron en envoyant *a_ ordre* en fonction de l'angle observé et de l'angle requis *a_ angle_ c*. Ainsi la commande de la gouverne aileron est implantée par la chaîne fonctionnelle $L_1 = (a_angle, a_angle_o, a_ordre)$. La commande sur la gouverne de profondeur fonctionne de la même manière.

Les lois de contrôle des ailerons et des gouvernes de profondeur communiquent au travers des fonctions *LPt* et *LPg* de façon à vérifier que les ordres envoyés sont consistants. Cette consolidation est néanmoins moins contrainte que la commande et supporte des latences plus longues. De ce fait les communications entre ces fonctions n'imposent pas de précédences strictes et une fonction peut consommer des données produites dans des instants moins récents. Néanmoins, les données de consolidation doivent être suffisamment fraîches. De ce fait, la chaîne $L_2 = (z_acc, z_acc_o, d_2, a_angle_c, a_ordre)$ tolère une latence maximale plus longue mais bornée.

La loi de guidage (*LG*) calcule une série d'accélération à appliquer pour atteindre la position consigne du pilote. Cette action est décrite par la chaîne $L_3 = (pos_c, r_lt_acc, a_angle_c, a_ordre)$ et est moins contrainte. On a vu figure 1.1 que la loi de guidage peut être appelée moins souvent.

Ces deux derniers exemples de chaîne illustrent que la spécification des communications n'est pas toujours déterministe : (1) quand les lois sont faiblement couplées ; (2) quand la latence globale est connue mais pas répercutée au niveau local. Permettre une telle liberté au concepteur simplifiera la conception, notamment quand le système est constitué de 5 000 fonctions et 20 000 communications ; et empêchera que les choix faits en amont ne rendent l'implantation sur la cible complexe, voire impossible.

Langage de spécification incomplète Puisque seuls les systèmes mono-périodiques sont considérés, un sous-ensemble de PRELUDE étendu avec (1) l'opérateur « don't care » et (2) la possibilité d'exprimer des contraintes de latence sur des chaînes fonctionnelles a été (re-)défini. A partir d'une telle spécification, le compilateur traduit la spécification partielle en un programme PRELUDE qui respecte les contraintes de latence et de causalité. La traduction est syntaxique dans le sens où chaque opérateur « don't care » est soit éliminé soit remplacé par un `fby` . Ce problème est équivalent à résoudre un ensemble de contraintes pseudo-booléennes.

La partie de la grammaire de PRELUDE à laquelle est ajouté l'opérateur *dc* et une liste d'expression de contraintes de latence est donnée dans la syntaxe de la figure 2.14.

La latence d'une chaîne en PRELUDE est exprimée par une durée, ex. 40 ms. Dans le cas restreint mono-périodique, une latence peut être exprimée de manière plus simple, à savoir comme le nombre d'instants logiques acceptables, en d'autres termes comme le nombre maximal de délais (`fby`) qui peuvent être insérés.

program	::=	*imported_node_list*;
		`node` $N(io)$ `returns` (io) [`var` io; *req_list*;] `let` *eq_list* `tel`
imported_node_list	::=	*imported node N(io) returns (io)*
		| *imported node N(io) returns (io)*; *imported_node_list*
io	::=	*id* | *io*, *id*
eq_list	::=	*eq* | *eq_list eq*
eq	::=	*io* = *exp*;
exp	::=	*id* | *cst* | $N(id, \ldots, id)$ | *cst* `fby` *id* | *cst* `dc` *id*
req_list	::=	*req* | *req req_list*
req	::=	*req (io)* $< k$;

FIGURE 2.14 – Syntaxe du langage de spécification partielle

Exemple 13. *Le système de commande de vol de la figure 2.13 s'écrit alors :*

```
node cdv_incomplet (a_angle, y_acc, e_angle, z_acc, pos_c)
returns (a_ordre,  e_ordre)

req (z_acc, z_acc_o, d2, dc2, a_angle_c, a_ordre) < 1; --L2
req (pos_c, r_lt_acc, dc6, a_angle_c, a_ordre) < 4; --L3

var e_angle_o, o_a_angle, r_a_angle, o_y_acc, r_e_angle, o_z_acc, lg_fb,
    r_lg_acc, d1, d2, r_lt_acc, lt_fb, dc1, dc2, dc3, dc4, dc5, dc6;
let
  a_angle_o = FAA(a_angle);                y_acc_o = ACA(y_acc);
  e_angle_o = EAF(e_angle);                z_acc_o = ACE(z_acc);
  a_ordre = LAA(a_angle_o,a_angle_c);    e_ordre = LAE(e_angle_o, a_angle_c);
  (lg_fb, d2, e_angle_c) = LPg(dc1, z_acc_o, dc4);
  (a_angle_c, lt_fb, d1) = LPt(dc2, y_acc_o, dc6);
  (r_lt_acc, r_lg_acc) = GL(dc3, dc5, p_ordre);

  dc1 = 0 dc d1;                         dc2 = 0 dc d2;
  dc3 = 1 dc lt_fb;                      dc4 = 0 dc r_lg_acc;
  dc5 = 1 dc lg_fb;                      dc6 = 0 dc r_lt_acc;
tel
```

On retrouve les contraintes de latence sur les deux chaînes L_2 et L_3. L_1 imposant des communications directes a été codée en dur, c'est-à-dire sans `fby` *ni* `dc`*. Il faut utiliser des variables intermédiaires (dc1 à dc6) pour exprimer les libertés de communication tout en respectant la syntaxe.*

La sémantique de Kahn de l'opérateur « don't care » est :

$$\texttt{dc}^{\#}(cst, s) \quad = \quad < \texttt{fby}^{\#}(cst, s) | s^{\#} >$$

Elle exprime que l'opérateur `dc` peut être remplacé soit par un `fby` soit par l'identité.

Exemple 14. *Nous illustrons la sémantique de l'opérateur* `dc` *sur un exemple simple.*

```
imported node plus (i1,i2) returns (o);
node ex (i) returns (o)
var v1, v2;
let
  v1 = 0 dc i;
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

i	5	3	7	...
v_1	$\{0, 5\}$	$\{5, 3\}$	$\{3, 7\}$	
v_2	$\{1\}$	$\{0, 5\}$	$\{5, 3\}$	
o	$\{1, 6\}$	$\{3, 5, 8, 10\}$	$\{6, 8, 10, 12\}$	

Concrétisation d'une spécification incomplète L'idée est donc de traduire une spécification incomplète en un programme PRELUDE, c'est-à-dire sans opérateur `dc` et sans latences exprimées dans `req` . La traduction est purement syntaxique et remplace chaque `dc` par une communication directe ou un `fby` .

Définition 6. *Un système (ou un programme) est dit* concret *si et seulement si il ne contient aucun opérateur* `dc` *et aucune exigence de latence* `req` *.*

Exemple 15. *Reprenons l'exemple simple 14. Il existe plusieurs versions concrètes de ce programme. Si on applique une traduction purement syntaxique, on obtient deux versions :* ex1 *où* `dc` *a été supprimé et* ex2 *où* `dc` *est devenu un* `fby` *. Si on acceptait d'autres types de traduction, on pourrait obtenir des programmes plus complexes qui alterneraient des communications directes ou non (comme proposé dans* ex3*).*

```
node ex1 (i) returns (o)
var v1, v2;
let
  v1 = i;
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

```
node ex2 (i) returns (o)
var v1, v2;
let
  v1 = 0 fby i;
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

```
node ex3 (i) returns(o)
var v1, v2, j;
let
  j = true fby (not j);
  v1 = if j then i
       else (0 fby i);
  v2 = 1 fby v1;
  o = plus(v1, v2);
tel
```

Soit sys le programme abstrait, $\texttt{dc}(sys) = \{dc_1, \dots, dc_n\}$ représente l'ensemble des opérateurs `dc` du système ordonné par leur ordre d'apparition. On note $p = (dc_i \mapsto \texttt{fby})sys$ le programme obtenu en substituant l'opérateur dc_i par `fby` dans sys. De la même façon, $(dc_i \mapsto id)sys$ représente la substitution de dc_i par l'identité et $(dc_x \mapsto op, \dots, dc_y \mapsto op)sys$ représente le programme p résultant de l'ensemble des substitutions $dc_x \mapsto op, \dots, dc_y \mapsto op$ (où $op \in \{\texttt{fby}, id\}$).

Définition 7 (Instance)**.** *Soit p un programme concret et sys un programme abstrait tel que* $\texttt{dc}\ (sys) = \{dc_1, \dots, dc_n\}$*. p est une* instance *de sys si et seulement si il existe un ensemble de substitutions $dc_1 \mapsto op_1, \dots, dc_n \mapsto op_n$ tel que :*

$$p = (dc_1 \mapsto op_1, \dots, dc_n \mapsto op_n)sys$$

Dans la suite, $sys[b_1, \dots, b_n]$ représente l'instance $p = (sub_1, \dots, sub_n)sys$ telle que :

$$\begin{cases} sub_i = dc_i \mapsto id & si\ b_i = 0 \\ sub_i = dc_i \mapsto \texttt{fby} & si\ b_i = 1 \end{cases}$$

Exemple 16. *Le programme* ex *de l'exemple 14 a deux instances montrées dans l'exemple 15 :* ex1=ex[0] *et* ex2=ex[1].

L'objectif est de générer des instances respectant les latences et la causalité.

Définition 8. *Soit sys un système abstrait et $L = (x_1, \dots, x_n)$ une chaîne fonctionnelle de sys. La latence de L est inductivement définie par :*

$$Lat_{sys}(x_1, \dots, x_n) = \begin{cases} 1 + Lat_{sys}(x_1, \dots, x_{n-1}) & si\ x_n \xleftarrow{\texttt{fby}} x_{n-1} \\ Lat_{sys}(x_1, \dots, x_{n-1}) & si\ x_n \overset{0}{\leftarrow} x_{n-1}\ ou\ x_n \xleftarrow{\texttt{dc}} x_{n-1} \end{cases}$$

avec $Lat_{sys}() = 0$ et $()$ la chaîne fonctionnelle vide.

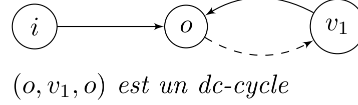
Exemple 17. *Pour le programme* ex *et la chaîne $L = (i, v_1, v_2, o)$, on $Lat_{ex}(L) = Lat_{ex1}(L) = 1$ et $Lat_{ex2}(L) = 2$.*

Définition 9. *La chaîne fonctionnelle* $\mathcal{C} = (x_1, \ldots, x_n, x_1)$ *est un* dc-cycle *si et seulement si :*

$$\exists i, 1 \leq i \leq n, x_i \overset{\text{dc}}{\longleftarrow} x_{(i \ mod \ n)+1}$$

Exemple 18. *Le programme suivant contient un dc-cyle.*

```
imported node n1 (i1 ,i2) returns (o);
node causal (i) returns (o)
var v1;
let
  o = n1(i,v1);
  v1 = 0 dc o;
tel
```



(o, v_1, o) *est un dc-cycle*

L'instance causal[0] *n'est pas causale tandis que* causal[1] *l'est.*

La recherche d'instances valides se traduit en un problème pseudo-booléen et le prototype développé par R. Wyss utilise le solveur SAT4J [LP10] pour trouver les solutions. Considérons un système abstrait sys, tel que `dc` $(sys) = \{dc_1, \ldots, dc_n\}$, et devant satisfaire la contrainte $Lat(\mathcal{C}) < k$ où $\mathcal{C} = (x_1, \ldots, x_n)$. Une instance $p = sys[b_1, \ldots, b_n]$ satisfait la contrainte de latence si et seulement si :

$$\sum_{j=1}^{l} b_{i_j} < k - Lat_{sys}(\mathcal{C})$$

Les variables du problème pseudo-booléen sont donc les b_{i_j}, tandis que $k - Lat_{sys}(\mathcal{C})$ est une constante.

Les contraintes de causalité s'expriment également en problème pseudo-booléen. On énumère les cycles élémentaires $\mathcal{C} = (x_{i_1}, \ldots, x_{i_m}, x_{i_1})$ du graphe de dépendances en utilisant un algorithme de recherche de cycle comme [Tar73]. Tout cycle $\mathcal{C}$ est acceptable si et seulement si :

$$\sum_{j=1}^{l} b_{i_j} + Lat_{sys}(\mathcal{C}) > 0$$

Exemple 19. *Considérons le programme suivant :*

```
node trad (i) returns (o1,o2)
req (i,v1,v2,v3,o2) < 3;
var v1, v2, v3;
let
  o1 = n1(i,v1);
  v1 = 0 dc o1; --dc1
  v2 = 0 dc v1; --dc2
  v3 = 0 fby v2;
  o2 = n2(v2);
tel
```

L'ensemble de contraintes est :

$$\begin{cases} b1 + b2 + 1 < 3 \\ b1 > 0 \end{cases}$$

Pour l'exemple des commandes de vol figure 2.13, avec $cdv_incomplet(dc1, dc2, dc3, dc4, dc5, dc6)$, il y a 4 instances valides : 101111, 101011, 101010, 101101. Les expérimentations menées ont prouvé qu'il était possible de traiter des spécifications de grandes tailles (le prototype a notamment généré des solutions pour une étude de cas composée de 3994 nœuds importés et 16186 variables).

2.5.3 Travaux connexes.

Latence de chaînes fonctionnelles Les latences et les propriétés de performance sont généralement étudiées au niveau de l'implantation. Trois types de latences sont généralement étudiés : les temps d'exécution pire cas (WCET) d'une tâche ou d'un thread sur un processeur donné [CPRS03 ; Abs], le temps de réponse pire cas (WCRT) d'une transaction (i.e., un ensemble de tâches formant une chaîne) sur un calculateur et pour une stratégie d'ordonnancement donnée [TCN00 ; LPT09 ; RGR09], et enfin les temps de traversée pire cas (WCTT) de réseaux temps réel (réseaux commutés, bus CAN, etc.) [LT01 ; Mar04].

Plusieurs travaux ont tenté de capturer la notion de latence d'un système global composé de composants logiciels, exécutés sur des processeurs et communiquant via un ou des réseaux temps réel. On

peut citer les travaux de F. Carcenac [Car05] et de M. Lauer [LEBP11a] qui proposent une méthode d'analyse de latence sur des systèmes avioniques. Cependant, comme précédemment, l'ensemble de ces travaux n'adressent eux aussi que le niveau implantation, soit donc un niveau tardif dans le processus de conception.

Les langages de description d'architecture comme AADL (*Architecture and Analysis Design Language*) [FGH06], MARTE (*Modeling and Analysis of Real-time and Embedded systems*) [Gro07] ou SYSML (*System Modeling Language*) [OMG10] permettent de spécifier de manière précise les motifs de communication entre composants fonctionnels. Les connexions immédiates imposent des contraintes du producteur vers le consommateur, ce qui n'est pas le cas les communications avec délais. Il est également possible d'imposer des latences fonctionnelles de bout en bout sur les « flows » (dans ce cas, un flow est un chemin à travers les sous-composants d'un composant).

Les auteurs de [OTS99] et [LS81] proposent une méthodologie pour optimiser les temps de calcul dans les circuits synchrones. A partir d'une spécification déterministe, la méthode, appelée *retiming*, récrit la spécification en un programme synchrone équivalent en insérant et supprimant des délais. Cependant ces travaux sont restreints à des systèmes mono-périodiques.

Spécifications incomplètes Les systèmes temps réel exécutés avec un ordonnancement standard sans gestion des communications reviennent à mettre des « don't care » sur toutes les communications. Quand une précédence est associée, il y a une communication directe (resp. indirecte) entre le prédécesseur et le successeur (resp. du successeur vers le prédécesseur).

Les auteurs de [MH96] ont défini un langage non déterministe basé sur les automates booléens I/O. La sémantique d'un tel programme est définie comme l'ensemble des programmes déterministes dont le comportement est inclus dans la version non déterministe. Leur langage a une expressivité plus grande que le nôtre. La traduction d'un programme non déterministe en un programme synchrone est faite par synchronisation avec un « oracle ». La façon de concevoir cet oracle est hors du sujet du papier.

Les langages de description d'architecture proposent des motifs complexes de communication déterministes. Le langage CCSL [MDAS10] utilisé dans MARTE est plus proche de nos travaux. Il est possible de spécifier des contraintes d'horloges, c'est-à-dire des contraintes entre les rythmes des différentes activations. A partir de ces contraintes, il est possible d'instancier plusieurs systèmes satisfaisant les contraintes.

Plus récemment, Haibo Zeng et al. sont partis d'une spécification PRELUDE partielle et ont réussi à instancier plusieurs paramètres dont les `fby` avec une approche MILP [AZDG13].

2.6 Conclusion

Le langage PRELUDE répond à un besoin réel de spécification d'assemblages multi-périodiques et a été défini de façon à s'insérer naturellement dans un processus de développement d'applications de type contrôle/commande. Cette spécification d'assemblage multi-périodique est déjà fournie par SIMULINK et il était logique d'y réfléchir également dans le paradigme synchrone. PRELUDE offre au concepteur une vue explicite de la manière dont les données sont consommées entre tâches contrairement à MATLAB/SIMULINK qui réalise un certain nombre de choix implicites, qu'il est parfois délicat de modifier. Le langage permet également de calculer des propriétés de latence ou de fraîcheur à ce niveau de description. La syntaxe étant très proche de LUSTRE, l'utilisateur n'a pas de réel effort d'apprentissage, si ce n'est la gestion des nouveaux opérateurs.

Les extensions plus récentes se rapprochent encore davantage des habitudes en automatique de ne spécifier que partiellement les consommations de données. Une fois le programme PRELUDE généré, l'utilisateur peut alors l'analyser, modifier les contraintes de haut niveau pour re-générer un autre résultat ou changer directement le programme pour imposer des contraintes supplémentaires.

L'ensemble de tâches généré s'inscrit dans les implantations temps réel standard. Cela ouvre donc des perspectives d'exécutions parallèles sur les plates-formes nouvelle génération. Plusieurs pistes seront explorées dans les prochains chapitres.

Chapitre 3

Modèles d'exécution prédictibles pour multi/pluri-cœurs

Le chapitre précédent décrivait le langage formel de description d'assemblages multi-périodiques Prelude. Une implantation possible du code généré sur une cible mono-processeur hébergeant un système d'exploitation temps réel Marte OS et ordonnancé selon la politique EDF pré-emptif a été montré dans la partie 2.3.4. L'objectif de ce chapitre est de présenter les travaux réalisés sur la définition de principe d'exécution « prédictible » de programmes Prelude sur des cibles multi/pluri-cœurs ainsi que l'environnement associé développé à l'ONERA depuis 2008, appelé SchedMCore.

3.1 Contexte

L'industrie aéronautique utilise des COTS (*off-the shelf component*, composant sur étagère) afin de réduire les coûts, les temps de mise sur le marché et faciliter la certification (puisqu'un composant grand public est supposé « *mâture* »). Les générations actuelles de processeurs COTS sont des multi-cœurs, c'est-à-dire des puces constituées d'un petit nombre de processeurs reliés par un bus partagé. Les nouvelles générations de processeur sont des pluri-cœurs, c'est-à-dire des puces constituées de nombreux processeurs reliés par un réseau sur puce. Ces deux types d'architectures offrent d'excellentes performances d'exécution en moyenne. Néanmoins leur embarquement pour des applications critiques est une problématique non résolue sur laquelle industriels et académiques travaillent depuis quelques années. En effet, les multi/pluri-cœurs ont des comportements dynamiques très compliqués à prédire car ils contiennent des mécanismes tels que plusieurs niveaux de cache ou des prédicteurs de branchement qui prennent des décisions par anticipation. De plus, plusieurs ressources sont partagées et les accès concurrents sur le médium de communication ou la mémoire introduisent des variations difficilement prédictibles dans les temps d'accès. De ce fait, la plupart des plates-formes multi/pluri-cœurs COTS ne sont pas « prédictibles » [WR12] dans le sens où calculer le pire temps d'exécution (WCET) avec une méthode d'analyse statique [WEE08] :

- — soit produit des résultats très pessimistes (pour chaque accès à une ressource, le calcul imagine le pire scénario où tous les cœurs accèdent à la ressource en même temps),
- — soit est impossible (s'il n'est pas possible d'associer un temps pire de traversée sur le bus par exemple).

3.1.1 Besoins et objectifs

L'objectif des étapes d'analyses temps réel et d'exécution prédictible sur multi/pluri-cœurs COTS est triple : (1) simuler fonctionnellement un programme Prelude ; (2) calculer les pires temps d'exécution et vérifier l'ordonnançabilité d'un programme Prelude et plus généralement d'un ensemble de tâches périodiques avec contraintes de précédence ; (3) exécuter le programme sur la cible.

Base de codage

Au début du travail de thèse de Mikel Cordovilla (2008-2012), il n'existait que peu d'outils d'analyse et d'exécutifs temps réel pour multi-cœurs. Cela tenait au fait que la plupart des résultats mono-processeurs ne s'étendent pas à ces cibles et de nouvelles approches devaient être envisagées. Ses travaux sont la

continuité naturelle du portage de PRELUDE sur un un système d'exploitation temps réel (RTOS - real-time operating system) et reposent sur plusieurs hypothèses fortes : (1) le WCET d'une tâche est calculable sur un multi-cœurs ; (2) les RTOS sont également prédictibles ; (3) le coût des migrations et des préemptions est facilement calculable. Ces hypothèses n'étant pas suffisamment réalistes, nous avons orienté par la suite nos travaux vers des principes plus prédictibles. Néanmoins, M. Cordovilla a posé les bases de codage de nos exécutifs et de l'environnement SCHEDMCORE. Le fondement de l'approche repose sur le paradigme de l'« *orienté par le temps* », c'est-à-dire que les décisions sont prises à des instants logiques. Ce choix s'est naturellement imposé du fait de la sémantique de PRELUDE et d'une meilleure maîtrise des comportements obtenus.

Principe des modèles d'exécution

Une solution pour forcer la prédictibilité sur les multi/pluri-cœurs de type COTS consiste à utiliser un *modèle d'exécution* adéquat [ABD13]. Un tel modèle est un ensemble de règles à suivre par le concepteur lors de l'implantation de façon à éviter ou au moins réduire les comportements non prédictibles.

Plusieurs projets et collaborations nous ont amené à définir trois modèles d'exécution reposant sur des séquencements partitionnés non pré-emptifs calculés hors-ligne et répondant à plusieurs règles de programmation.

Modèle d'exécution 1

Nos travaux ont débuté avec Airbus en 2007 dans le cadre du projet industriel MARTIAC/SHRINK (2007-2012). Le modèle avait initialement été défini pour une architecture multi-processeurs reliés par un réseau haut débit [BHP08]. Le modèle a ensuite été étendu pour des multi-cœurs (similaires aux architectures Freescale MPC8641D ou P4080) et a donné lieu à un brevet porté par Airbus [JTA12]. La première publication officielle s'est faite dans [BCNP12].

Le modèle proposé est en adéquation avec les applications de type contrôle/commande, en effet ces dernières sont des ensembles de tâches multi-périodiques souvent géométriques (les périodes sont multiples entre elles) : il est donc aisé de calculé des ordonnancements hors-lignes non pré-emptifs. De plus, ces applications sont composées de beaucoup de nœuds de petites tailles, ce qui facilite le placement dans des zones mémoires de petites tailles et des fenêtres d'exécution de petites tailles. Les données manipulées peuvent facilement être découpées si besoin.

Modèle d'exécution 2

Nous avons également travaillé avec Thales dans le cadre du projet industriel WC$\star$T (2011-2015) dont l'objectif était de porter une application avionique sur un multi-cœurs COTS en assurant la prédictibilité. Ce deuxième type d'application avionique est une version simplifiée mais représentative du *Flight Management System (FMS)* développée par Thales. La version originale est le FMS220 [THA10], embarqué dans les avions régionaux (ATR-72, ATR-42) et sur certains hélicoptères (Sikorsky S-76D). Dans les avions modernes, le FMS (1) aide l'équipage à naviguer de façon à optimiser les trajectoires en réduisant la consommation de carburant ; (2) fournit des informations de la situation géographique, comme la liste des aéroports les plus proches. Le pilote et le co-pilote interviennent dans les calculs du FMS et peuvent forcer des trajectoires, des positions ou simplement le désactiver. La version originale était séquentielle et le code a été récrit sous forme multi-tâches de façon à permettre sa parallélisation [Gir12a ; Gir12b].

Cette application est moins régulière qu'une application de type contrôle/commande puisqu'elle est constituée de tâches indépendantes périodiques et apériodiques (nécessaires pour tenir compte des requêtes du pilote). Le multi-cœurs cible est également différent puisqu'il s'agit du TMS de Texas Instrument. Le modèle proposé a donc été conçu en fonction de ces contraintes.

Modèle d'exécution 3

Le dernier modèle résulte à la fois d'un projet interne ONERA PR SCC (2011-2013) dont l'objectif était d'étudier l'embarquabilité du pluri-cœurs SCC (Single-chip Cloud Computer) et des travaux dans le cadre du projet RTRA TOAST (Time-Oriented criticAl SysTems [1] - partenaires IRIT, ISAE, LAAS, ONERA, Thales - 2011-2015) finançant notre post-doctorant Wolfgang Puffitsch.

1. http://www.irit.fr/torrents/toast/toast.php

Le modèle est dédié aux pluri-cœurs et prend en compte les aspects réseau sur puce. Les applications considérées sont de type contrôle/commande et proviennent de programmes PRELUDE (ou similaires).

3.1.2 Travaux connexes

Sur l'implantation d'ordonnancements classiques

L'analyse d'ordonnançabilité sur multi-processeurs est nettement plus complexe que celle sur mono-processeur [DB11]. Cela tient à plusieurs facteurs, notamment :

— la non robustesse des politiques dans le sens où un ensemble de tâches ordonnançable avec une politique d'ordonnancement peut devenir non ordonnançable lorsqu'on réduit sa charge sur le système (en diminuant les pires temps d'exécution ou en augmentant les échéances) [Gra69] ;

— le scénario pire cas (ou *instant critique*) n'est pas le démarrage synchrone de toutes les tâches [LMM98] et est difficile à trouver ;

— la manque de connaissance sur l'intervalle de faisabilité. Cela a un impact sur les techniques basées sur l'exploration exhaustive des comportements. En effet, il faut visiter tous les états accessibles et donc déterminer une condition de finitude. La question est simple dans le cas mono-processeur [LM80] puisqu'il s'agit de $[0, H]$ pour les tâches synchrones et $[0, \max O_i + 2H]$ pour les tâches asynchrones, où H est l'hyper-période de l'ensemble de tâches périodiques à savoir $H = ppcm(T_i)$. Dans le cas multi-processeurs, Cucu et Goossens [CG11] ont montré que (1) l'intervalle de faisabilité est $[0, H]$ pour un ensemble de tâches synchrones à échéances contraintes, (2) cet intervalle peut être calculé inductivement dans le cas d'un ordonnanceur à priorité fixe. Grolleau et al. [GGC13] ont proposé une nouvelle borne :

$$\prod_i (max(0, (O_i + D_i - T_i)) + 1)H$$

De ce fait, les conditions suffisantes d'ordonnançabilité sont pessimistes. Nous avons donc choisi de développer un analyseur d'ordonnançabilité à base d'exploration d'automates, dans la veine des travaux de [AFM02 ; BB07 ; GGD07 ; GGL08 ; DILS10]. Des travaux plus récents [LGG11 ; GGL13] utilisent des idées similaires aux nôtres pour des tâches sporadiques.

Les exécutifs disponibles au démarrage de la thèse étaient également peu nombreux et le plus abouti était LITMUS [CLB06]. Nous sommes repartis des idées du Meta-Scheduler [LRSF04] pour développer (rapidement) un exécutif le plus indépendant possible de la cible permettant la simulation de programmes PRELUDE.

Sur les modèles d'exécution

La première solution pour contrer les problèmes de prédictibilité consiste à ne pas utiliser un COTS mais à développer une architecture ad hoc prédictible. C'est la voie choisie par les projets PRET [LRL10], MERASA [UCS10] ou T-CREST [Spa12] par exemple. Dans le domaine avionique, les tendances actuelles sont plutôt d'embarquer des COTS et donc de chercher des moyens de les rendre certifiables.

L'utilisation de modèle d'exécution n'est pas quelque chose de nouveau : le modèle *Bulk Synchronous Parallel (BSP)* [CFSV95 ; ST98] avait été défini pour restreindre les comportements de programmes parallèles sur multi-processeurs afin d'en évaluer les performances. L'idée principale de BSP est de restreindre le modèle de programmation en une séquence de *supersteps* où un superstep se déroule en même temps sur tous les processeurs et se fait en trois phases :

1. une étape de calcul locale à chaque processeur,
2. une étape de communication globale de données entre processeurs,
3. une étape de synchronisation (une barrière).

D'autres chercheurs ont travaillé sur le même sujet en même temps que nous. Une des premières publications sur le sujet est [CRM10a] où les auteurs considèrent une architecture multi-cœurs avec un L2 partagé et un accès time division multiple access (TDMA) sur le bus interne. Ils montrent alors qu'une analyse de WCET est possible. Dans le même ordre d'idée, les auteurs de [SCT10] supposent un accès TDMA sur le bus interne. Ils imposent de plus une spécification applicative de la forme : un processus est décomposé en *super-blocs* qui ne peuvent être pré-emptés. Un super-bloc regroupe 3 étapes : acquisition, exécution et écriture (on reconnaît les super-steps de BSP). Ces travaux supposent qu'il est possible de configurer le protocole d'accès au bus interne, ce qui n'est en général pas le cas des COTS.

Le modèle d'exécution prédictible PREM [PBB11] a été défini pour architecture mono-processeur afin d'empêcher les interférences entre les calculs CPU et les entrées/sorties (I/O). A nouveau le système est un ensemble de tâches périodiques, chacune étant programmée comme un super-bloc : cette fois, il n'y a que deux phases exécution et communication. Pendant la phase d'exécution, aucun accès aux ressources partagées n'est toléré, c'est-à-dire que les codes et données sont stockés dans les caches. Durant une phase de communication, les périphériques sont bloqués (par un FPGA spécifique) et seule la tâche sur le CPU accède à la mémoire. Pour utiliser PREM, l'utilisateur doit annoter son code pour préciser les phases. Les auteurs ont étendu leur modèle pour des architectures multi-cœurs en séquençant les accès à la mémoire [BBP13].

3.2 SchedMCore : environnement de simulation/exécution de systèmes multi-périodiques

En complément aux travaux de J. Forget, Mikel Cordovilla [Cor12] a réalisé un environnement pour simuler et exécuter des programmes Prelude sur des architectures multi/pluri-cœurs. L'idée était de poursuivre le portage de Prelude comme cela avait était fait en mono-processeur avec Marte OS (voir partie 2.3.4). Une hypothèse fondamentale était donc d'utiliser RTOS.

Approche proposée

La figure 3.1 illustre l'approche proposée pour l'analyse et l'exécution d'une spécification Prelude sur une cible multi-cœurs.

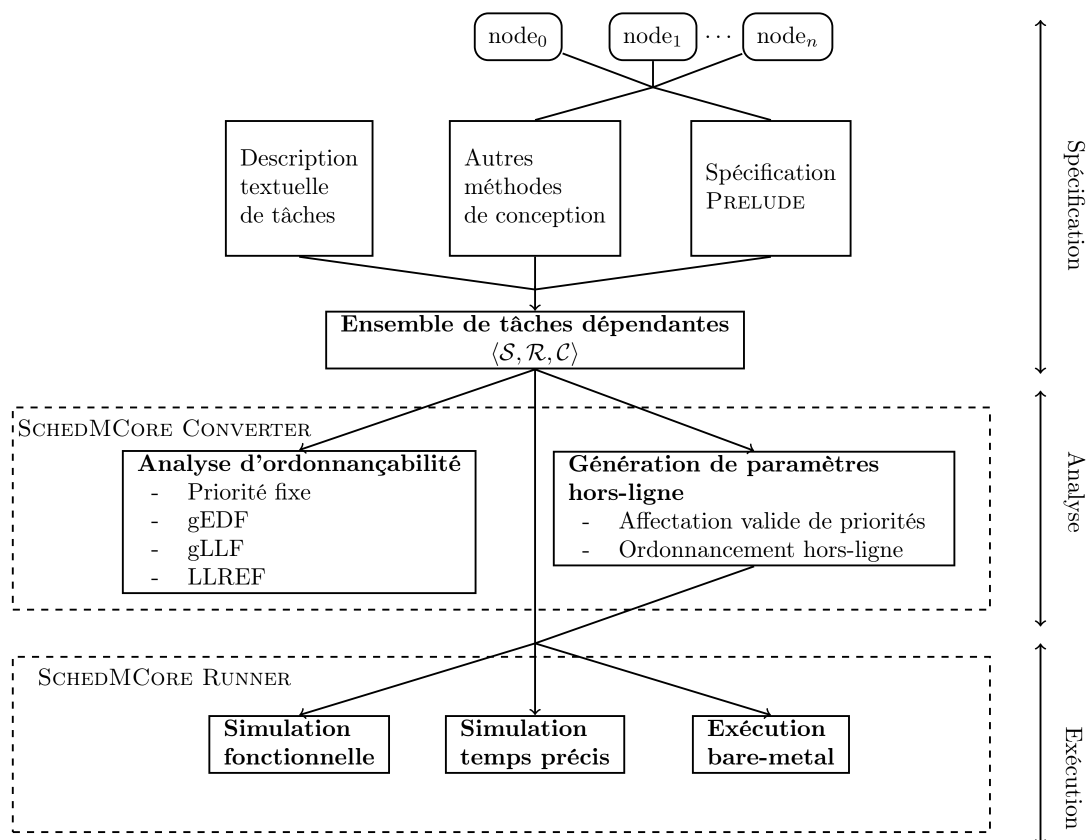


Figure 3.1 – Environnement Prelude-SchedMCore

La phase de spécification a été décrite dans le chapitre 2. Le point d'entrée de l'environnement est un ensemble de tâches périodiques communicantes reliées par des contraintes de précédence étendue $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$ (cf. partie 2.3.3). SchedMCore est divisé en deux branches :

— CONVERTER porte sur les analyses réalisées hors-ligne. Une première série concerne l'analyse d'ordonnançabilité de politiques en-lignes, décrite dans [CBNP11]. La deuxième série concerne la génération de priorités fixes valides ou d'un séquenceur hors-ligne, décrite dans [BBC12] ;
— RUNNER est l'exécutif qui permet de réaliser des simulations fonctionnelles ou temps réel précis, comme cela a été décrit dans [CBF11].

Les deux parties de SCHEDMCORE reposent sur une représentation commune (et une base commune de codage) à base d'un automate de configuration. Durant la thèse de Mikel Cordovilla, les politiques considérées étaient globales pré-emptives. Le codage a été étendu durant le post-doctorat de Wolfgang Puffitsch à des politiques partitionnées et/ou non pré-emptives. L'environnement est disponible à l'url `http://sites.onera.fr/schedmcore/`.

3.2.1 Encodage de l'évolution des tâches

Pour avoir une explication détaillée de SCHEDMCORE, le lecteur est invité à lire la thèse de Mikel [Cor12]. Nous rappelons ici les idées principales. Ce codage reprend la notion de configuration à l'instant t, de [CG06 ; BB07 ; GGL08], représentant le temps restant des paramètres 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.

Définition 10 (Configuration d'une tâche)**.** *Toute tâche τ_i à un instant t est caractérisée par le n-uplet :*

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

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.*

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

— $T_i(t) : \mathbb{N} \to [0, T_i]$ *est une fonction qui à un instant t représente le temps restant jusqu'au réveil du prochain job.*

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

— $D_i(t) : \mathbb{N} \to [0, D_i]$ *est une fonction qui à un instant t représente le temps restant jusqu'à l'échéance du job courant.*

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

— $C_i(t) : \mathbb{N} \to [0, C_i]$ *est une fonction qui à un instant t représente le temps d'exécution restant du job en cours d'exécution. On note $P(t)$ l'ensemble des tâches les plus prioritaires qui accèdent à un processeur à l'instant t.*

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

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

Définition 11 (État du système)**.** *Pour tout ensemble de tâches $\mathcal{S} = \{\tau_i\}_{i=1,..,n}$, l'état du système* conf(t) *à un instant t est défini par* $\text{conf(t)} = \langle \text{conf}(\tau_1, t), \ldots, \text{conf}(\tau_n, t) \rangle$.

Une exécution est alors une succession infinie de configurations.

Exemple 20. *Considérons l'ensemble de tâches $\{\tau_0 = (0, 5, 5, 2), \tau_1 = (1, 5, 5, 5), \tau_2 = (1, 5, 5, 2)\}$ à exécuter sur une architecture à deux processeurs selon une politique globale pré-emptive à priorité fixe avec* $\text{prio}(\tau_0) < \text{prio}(\tau_1) < \text{prio}(\tau_2)$. *Le calcul des configurations est donné ci-dessous et l'exécution est dessinée dans la figure 3.2.*

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)	$\{0\}$	$\{0,1\}$	$\{1,2\}$	$\{1,2\}$	$\{1\}$	$\{0,1\}$	$\{0,1\}$	...

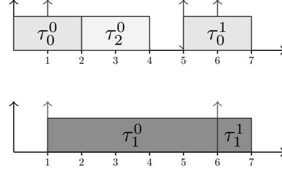


FIGURE 3.2 – Diagramme de Gantt de l'exécution de l'exemple 20

La nouveauté par rapport aux autres codages est la gestion des contraintes de précédence étendue, introduit dans la définition 3 p. 17. Soit deux tâches telles que $\tau_i \xrightarrow{M_{i,j},L_{i,j}} \tau_j$. 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 (n,n') tels que $\tau_{i.n} \to \tau_{j.n'}$, $n \in [0,(ppcm(T_i,T_j) \times L_{i,j})/T_i - 1]$ et $n' \in [0,(ppcm(T_i,T_j) \times L_{i,j})/T_j - 1]$.

Définition 12 (Contrainte de précédence à un instant t)**.** *L'état de la contrainte de précédence de $\tau_i \xrightarrow{M_{i,j},L_{i,j}} \tau_j$ à l'instant t est défini comme l'ensemble des mots actifs $M_{i,j}(t) = \{(m_1,m'_1),\ldots,(m_{k_t},m'_{k_t})\}$. Puisque le raisonnement est fait en relatif, cela induit que $M_{i,j}(t) \subseteq M_{i,j}$. On a $M_{i,j}(0) = M_{i,j}$.*

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

Exemple 21. *L'exécution de l'ensemble de tâches suivant*

	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*

$\mathcal{R}$

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

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

avec une politique d'ordonnancement globale pré-emptive à priorité fixe sur une architecture à deux processeurs est décrite dans le tableau suivant :

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

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

Le diagramme de Gantt est donné figure 3.3.

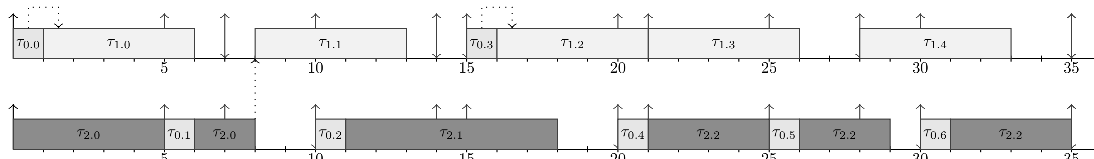


FIGURE 3.3 – Exécution sur deux processeurs et FP

3.2.2 Fenêtre de faisabilité

Les explorations menées sont finies car le nombre d'états est fini. En effet, les exécutions considérées ont lieu à des changements d'états à date entière et tous les paramètres sont exprimés en temps relatif. Ce résultat est publié dans [BBC12].

Théorème 1. *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* $p = \prod_i (C_i + 1)$.

Ainsi, CONVERTER génère

1. des codes C ou UPPAAL pour l'analyse d'ordonnançabilité de politiques en-ligne ainsi que pour la génération de priorités valides ;
2. des modèles UPPAAL pour la génération de séquenceur.

En effet, dans le premier cas, une fenêtre (pessimiste) est connue et l'algorithme avance dans le temps en sauvegardant les valeurs des $C_i(t)$ à chaque $t = \max_{i\leq n}(O_i) \mod (H)$. Lorsqu'un état visité a déjà été exploré, l'algorithme est assuré d'avoir tout analysé. Dans le deuxième cas, il faut faire un parcours plus intelligent avec backtrack d'où l'utilisation d'un model checker. Une branche de génération de modèles FIACRE [BBF08] a été codée avec Alexandre Hamez et Bernard Berthomieu.

Comparaison avec des bornes récentes Reprenons la borne proposée par Grolleau et al. [GGC13]

$$\prod_i (\max(0, (O_i + D_i - T_i)) + 1)H$$

Les deux valeurs sont incomparables. En effet, les C_i n'interviennent pas dans leur calcul tandis que le coût des dates de réveil est cher.

Exemple 22. *Dans certains cas, leur borne est meilleure. Soit* $\mathcal{S} = \{\tau_i = (1, 10, 10, 5)\}_{i\leq n}$. *Notre formule donne* $\max O_i = 1$, $H = 10$ *et* $\prod(C_i+1)*H = 6^n*10$. *L'autre formule donne* $\prod((O_i+D_i-T_i)_0+1)\times H = (\prod 2) * 10 = 2^n * 10$.

Dans d'autres cas, lorsque toutes les tâches ont des dates de réveil et que les C_i *sont petits, nous avons de meilleurs résultats. Soit* $\mathcal{S} = \{\tau_i = (10, 10, 10, 1)\}_{i\leq n}$. *Notre formule donne* $\max O_i = 10$, $H = 10$ *et* $\prod(C_i + 1) * H = 2^n * 10$. *L'autre formule donne* $\prod((O_i + D_i - T_i)_0 + 1) \times H = (\prod 10) * 10 = 10^n * 10$.

Enfin, les auteurs de [NYG13] ont proposé une borne meilleure que la nôtre dans tous les cas :

$$\max(O_i) + (1 + \sum_i C_i)H$$

Leur idée est de montrer que $C_i(t+H) \leq C_i(t)$ après $\max(O_i)$ et donc que le nombre de combinaison est plus faible.

3.3 Modèle d'exécution 1 : par tranches fines

Le premier modèle proposé a été présenté dans [BCNP12]. L'architecture cible est un multi-cœurs avec des caches L1 et L2 privés, comme un Freescale MPC8641D [Fre08] ou un Freescale P4080 [Fre10]. Notre solution est la plus poussée au niveau de la séparation des préoccupations temporelles par rapport aux autres modèles (comme BSP ou PREM). Le modèle d'exécution par tranches fines opère comme suit :

1. deux types de tranche alternent indéfiniment sur chaque cœur :
   - *tranches d'exécution :* un cœur ne peut exécuter que du code fonctionnel sans accéder aux ressources partagées. Les instructions et les données sont donc stockées dans les caches ;
   - *tranches de communication :* durant cette phase, le cœur ne fait aucun traitement fonctionnel. Il commence par écrire (*flusher*) ses données en cache dans la RAM, puis pré-charger dans les caches (*fetcher*) les prochaines instructions et données nécessaires pour la prochaine tranche d'exécution.
2. un *séquenceur synchrone statique* des tranches sur chaque cœur est défini hors-ligne. Il décrit un motif fini qui se répète à l'infini tel que les tranches de communication sont synchrones. Un exemple est donné dans la figure 3.4. Les boîtes blanches représentent les tranches d'exécution. Les blocs gris (resp. noirs) représentent les pré-chargements (resp. les écritures). Il faut donc que les cœurs soient parfaitement synchronisés et cette synchronisation est fournie par construction sur les processeurs Freescale mentionnés plus haut.

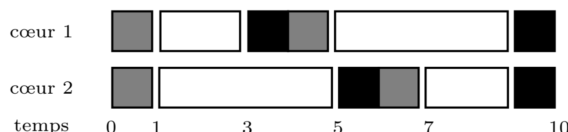


FIGURE 3.4 – Modèle d'exécution à tranche fine

Nous dirons dans la suite qu'un processeur COTS soumis au modèle d'exécution 1 est une *architecture tranchée.*

3.3.1 Validation du modèle

Pour ce premier modèle, nous avons montré qu'il était possible de calculer les pires temps d'exécution d'ensembles de tâches périodiques dépendantes (même modèle que celui issu de PRELUDE). La cible choisie était le Freescale MPC8641D, c'est-à-dire un dual core composé de deux cœurs Power PC e600 [Fre06], un *MPX Coherency Module (MCM)* qui sérialise les requêtes d'accès aux contrôleurs mémoire et la DDR. La figure 3.5 montre le processus de développement pour implanter une spécification PRELUDE sur le modèle d'exécution 1. Le processus est le suivant :

- les entrées sont la spécification fonctionnelle et le séquenceur synchrone choisi par l'intégrateur,
- (étape 1) consiste à évaluer le WCET de chaque tranche d'exécution séparément. Grâce aux hypothèses de confinement, cela se ramène à calculer un WCET pour un code séquentiel non préemptif sur un processeur unique sans comportement non prédictible, ce qui est un problème connu et résolu. Nous avons modélisé l'architecture en OTAWA avec l'aide d'Hugues Cassé pour calculer les WCET dans les tranches d'exécution.
- (étape 2) est la génération d'un placement statique des tâches et des communications sur l'architecture tranchée. L'algorithme calcule également les adresses mémoires *abstraites* des instructions et des données, de sorte que toutes les contraintes fonctionnelles (ex. précédences, périodes, échéances)

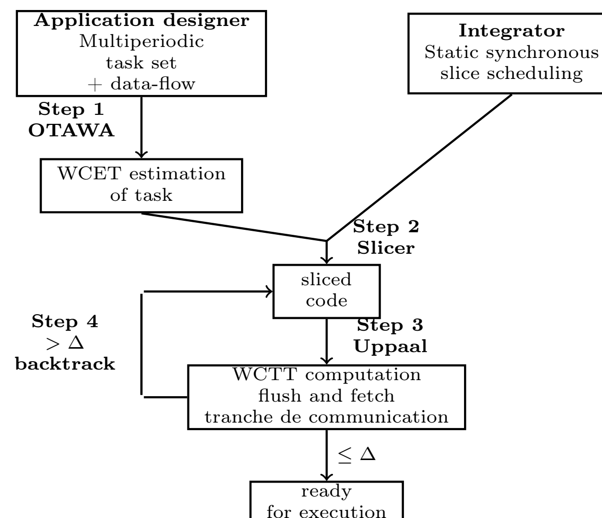


FIGURE 3.5 – Processus de développement

et de capacité (ex. taille des tranches, des caches) soient respectées. L'algorithme de placement est à base de programmation par contraintes sera abordé dans la partie 4.2.

— (étape 3) consiste à évaluer les pires temps de communication réseau (*worst case traversal time* - WCTT) de chaque tranche de communication et montrer qu'ils tiennent dans la longueur de la tranche (Δ). La solution choisie est d'explorer toutes les combinaisons d'écriture parallèle (resp. lecture) et des accès aux périphériques dans une tranche de communication en faisant un modèle du bus partagé, du contrôleur mémoire et de la RAM. Nous avons utilisé UPPAAL [BDL06] pour calculer les WCTT.

Nous avons appliqué la méthodologie à l'étude de cas avionique que nous avions présentée dans [BHP08]. Lorsque le calcul de WCTT échoue car le temps de communication dépasse la longueur (Δ), il faut alors proposer un autre placement ou un autre découpage des tâches. Cet aspect n'a pas encore traité dans nos travaux.

3.3.2 Calcul du WCET d'une tranche d'exécution

Le calcul statique d'un WCET passe par trois étapes principales [WEE08 ; Roc11] :

1. Construction du graphe de flot de contrôle (Control flow graph CFG) à partir du binaire du programme. Les nœuds sont des ensembles d'instruction (blocs de base) et les transitions relient ces blocs pour exprimer la structure comme les conditionnelles ou les boucles. Cette étape est automatique, le concepteur doit néanmoins fournir certaines informations comme les bornes des boucles ;
2. Analyse micro-architecturale : l'objectif est de calculer les temps pires d'exécution des blocs de base formant le CFG. Pour cela, il faut décrire finement le comportement temporel de chaque instruction dans le pipeline, les ressources du pipeline et prendre en compte les dépendances entre instruction. La construction du modèle bas niveau se fait à partir des documentations et des mesures.
3. Calcul du WCET : est obtenu en combinant les WCET des blocs de base. Une méthode possible est basée sur la résolution d'un ensemble de contraintes entières linéaires.

L'outil OTAWA [BCRS10] a été appliqué sur l'architecture tranchée. Il a donc fallu commencer par réaliser le modèle micro-architectural du PowerPC e600 [Fre06]. Les hypothèses reflétant une tranche d'exécution pour la modélisation sont les suivantes :

1. toutes les données sont pré-chargées dans le cache L1D et un accès coûte un temps constant,
2. toutes les instructions sont pré-chargées dans le cache L2 et un accès coûte un temps constant,
3. le prédicteur de branchement est désactivé (seul un prédicteur de branchement statique est accepté car déterministe).

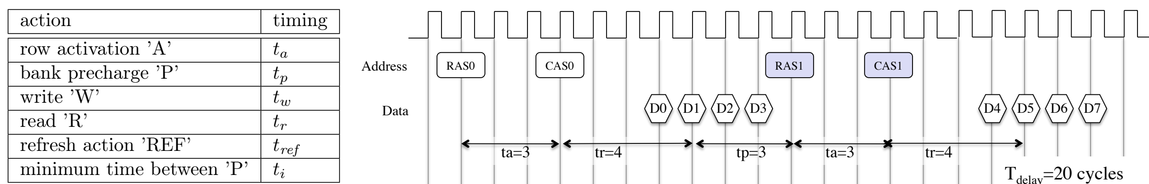
Ces hypothèses font que le L2 peut être vu comme une mémoire scratch-pad (SPM) avec un temps d'accès constant, le L1I n'est jamais utilisé et le L1D est un cache associatif 8 voies. Le WCET de chaque tâche de l'étude de cas a été évalué en Otawa [BHP08] en moins de 2 heures. A cette étape, il n'est pas nécessaire de connaître les adresses mémoires.

3.3.3 Calcul du WCTT d'une tranche de communication

Une fois un placement de l'application calculé, on connaît exactement les données qui doivent être écrites et pré-chargées dans chaque tranche de communication. L'objectif est de calculer le pire temps pour réaliser tous les flushes (resp. fetches) dans une tranche et montrer que cette valeur est plus petite que le temps alloué à la tranche. Il faut appliquer une approche similaire à celle du calcul de WCET : il faut décrire un modèle micro-architectural du comportement des accès mémoire incluant les cœurs, le bus interne, le contrôleur mémoire et la RAM.

Description des comportements en lecture et écriture

Une *référence* (ou référence mémoire, memory reference en anglais) représente une requête générée par un cœur, telle que *load* ou *store* d'une adresse mémoire. Une mémoire RAM [RDK00] est un composant de stockage tri-dimensionel organisé en *banks* (banques), *rows* (ou *memory pages*, lignes ou pages) et *columns* (colonnes). Une référence peut alors être vue comme un triplet (num bank, num row, num column).



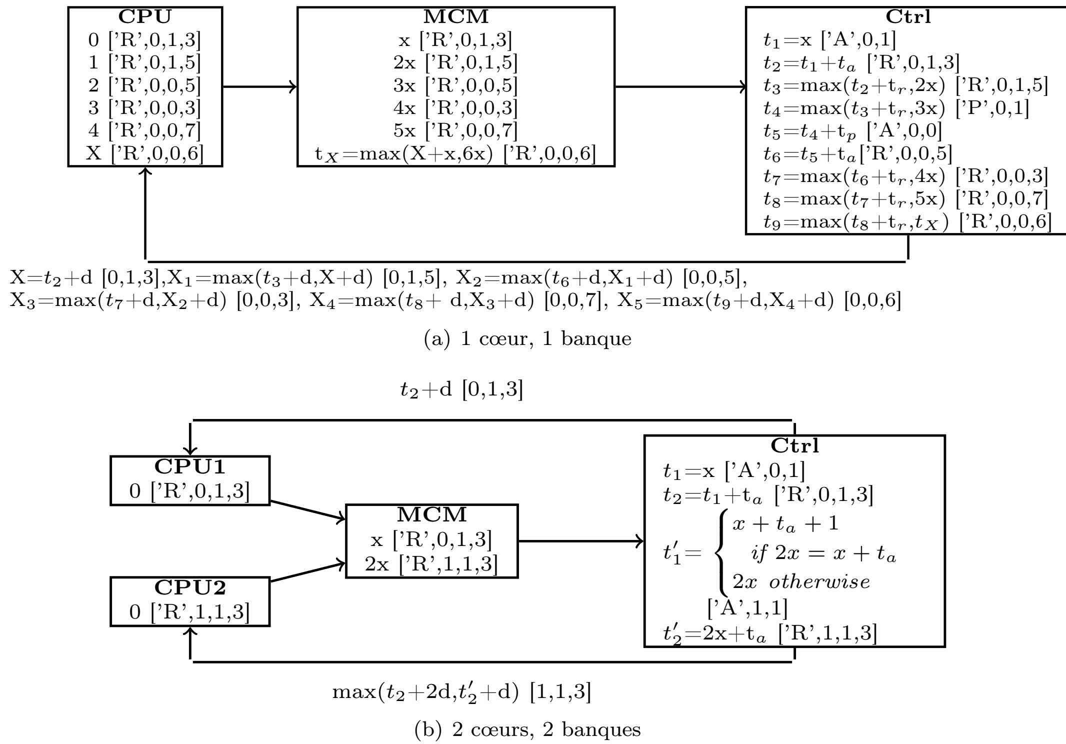
action	timing
row activation 'A'	t_a
bank precharge 'P'	t_p
write 'W'	t_w
read 'R'	t_r
refresh action 'REF'	t_{ref}
minimum time between 'P'	t_i

Figure 3.6 – Description des accès mémoire

Le comportement temporel de la mémoire peut être représenté comme illustré dans la figure 3.6. Les actions se déroulent dans un ordre précis : initialement la banque est *idle*. Ensuite, si une référence est placée dans le contrôleur, le contrôleur demande d'abord une activation A et il faut t_a cycles à la RAM pour stocker la ligne dans le buffer. Ensuite, le contrôleur émet la commande R ou W qui prend t_r ou t_w cycles à la RAM. Si la seconde référence accède à la même banque et la même ligne, alors le contrôleur demande directement une lecture ou une écriture. Mais s'il faut accéder à une autre ligne, le contrôleur demande d'abord un pré-chargement P qui prend t_p cycles, puis une activation et finalement la commande. Il y a une contrainte entre deux pré-chargements successifs : ils doivent être au moins séparés par t_i cycles. Il y a également un prix à payer pour passer d'une lecture à une écriture et vice versa. Mais dans notre cas, on ne fait soit que des écritures, soit que des lectures. Les banques travaillent indépendamment et stockent chacune leur ligne dans leur buffer local.

Les actions de *refresh* arrivent régulièrement sur la DDR pour des raisons physiques. Ces actions arrivent nb_{ref} fois pendant une durée de I_{ref}. Quand un refresh a lieu, toutes les banques se mettent en pré-chargement, font le rafraîchissement et se remettent en état *idle*.

Le comportement global d'accès à la mémoire est donné dans la figure 3.7 pour les requêtes en lecture. Les cœurs ont des FIFO de taille 5 (resp. 8) pour émettre les requêtes en lecture (resp. écriture). Le médium de communication, appelé le *MCM*, contient : (1) deux FIFOs de taille 8 pour stocker les requêtes de chaque cœur ; (2) une FIFO de taille 16 pour les données échangées avec la RAM. La taille de la FIFO du contrôleur mémoire est 4. Dans la figure 3.7(a), le cœur émet 6 requêtes : comme la FIFO est de taille 5, les 5 premières requêtes sont émises en séquence et la $6^{ème}$ doit attendre le traitement de la première pour être émise à son tour. Le cœur émet la requête ['R',0,1,3] (accès à banque : 0, numéro ligne : 1, numéro colonne : 3) à la date 0. Le cœur émet des requêtes à chaque cycle tandis que le taux de retransmission du MCM est x. La première requête arrive à 0 dans la FIFO du MCM et est transmise au contrôleur à x, la deuxième requête est reçue à 1 puis est traitée à x et donc retransmise à $x + x$. Le contrôleur traite la première requête reçue à x, comme la banque est *idle*, il demande une activation 'A' et après t_a demande une lecture 'R'. La seconde requête est disponible dans le contrôleur à $2x$ et le contrôleur peut commencer le traitement à $t_2 + t_r$. Ainsi, la lecture 'R' est lancée à $t_3 = \max(t_2 + t_r, 2x)$. La valeur de la première requête ['R',0,1,3] est envoyée au CPU et arrive à $X = t_2 + d$ où d est le taux



(a) 1 cœur, 1 banque

(b) 2 cœurs, 2 banques

FIGURE 3.7 – Comportement temporel en lecture

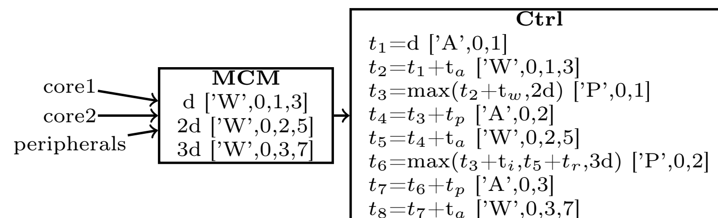
d'émission sur le bus de données. La $6^{ème}$ requête est transmise au MCM après traitement de la première requête X et après l'envoi de la $5^{ème}$, donc à $t_x = \max(X + x, 6x)$. La valeur de ['R',0,1,5] est donc émise après la transmission de la donnée précédente à X et après le travail de la RAM à t_3. Elle est donc reçue par le cœur à $\max(X + d, t_3 + d)$.

Lorsque plusieurs cœurs accèdent à la RAM, plusieurs entrelacements sont possibles. Les requêtes sont d'abord sérialisées par le MCM. Il y a donc plusieurs scénarios à l'arrivée sur le contrôleur :

— les cœurs accèdent à la même banque et génèrent des contentions en fonction du nombre de lignes qui peuvent être ouvertes simultanément ;
— chaque cœur accède à une banque différente et donc plusieurs lignes sont ouvertes en même temps. On réduit ainsi le nombre de 'P' et 'A', et donc la latence globale ;
— les cœurs accèdent à des contrôleurs différents.

La figure 3.7(b) décrit les entrelacements des requêtes en lecture émises par deux cœurs accédant à 2 banques distinctes. Dans la figure, la requête du cœur 1 ['R',0,1,3] est reçue en premier dans le MCM. Cette requête est ensuite traitée comme avant : une activation suivie d'une lecture puis émission sur le bus de données vers le cœur 1. La requête ['R',1,1,3] est traitée immédiatement par le contrôleur si elle n'est pas en conflit temporel avec une demande d'activation ou de lecture pour la première requête. Sinon, il faut attendre 1 cycle pour demander l'activation. Ensuite, il y a la lecture et l'envoi sur le bus.

Le cas des rafales d'écriture est plus simple à modéliser car il n'y a pas d'attente de retour depuis la RAM (plus exactement, le temps d'acquittement est caché par la lenteur d'émission du bus de données comparée au bus de requête). Si plusieurs composants (cœurs et périphériques) écrivent leurs données dans la même banque, alors le contrôleur passera la plupart de son temps à pré-charger des lignes :



Si en revanche les accès se font sur 3 lignes ouvertes simultanément alors on observe le comportement suivant :

MCM		Ctrl
12 ['W',0,1,3] 24 ['W',0,2,5] 36 ['W',0,3,7]	→	t_1=12 ['A',0,1], t_2=20 ['W',0,1,3] t_3=24 ['A',0,2], t_4=32 ['W',0,2,5] t_5=36 ['A',0,3], t_8=44 ['W',0,3,7]

MCM		Ctrl
12 ['W',0,1,3] 24 ['W',1,2,5] 36 ['W',2,3,7]	→	t_1=12 ['A',0,1], t_2=20 ['W',0,1,3] t_3=24 ['A',1,2], t_4=32 ['W',1,2,5] t_5=36 ['A',2,3], t_8=44 ['W',2,3,7]

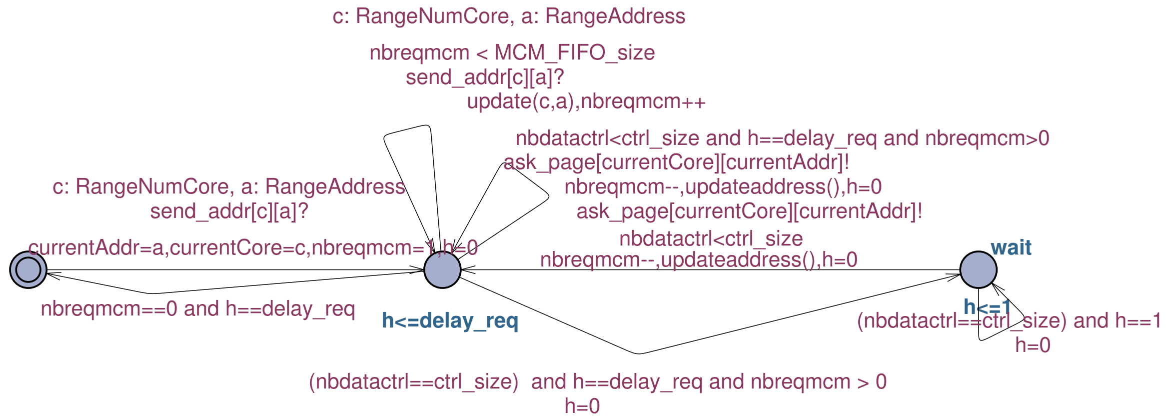
Calcul du temps pire cas

Il faut donc calculer le temps nécessaire pendant les tranches de communication pour écrire les données produites en RAM ainsi que le temps nécessaire pour pré-charger les données en RAM pour la tranche d'exécution suivante. Une première sur-approximation consiste à calculer le pire temps pour lire une donnée $t^{\max}_{read}$ (resp. écrire une donnée $t^{\max}_{write}$) ; puis multiplier par le nombre de lectures (resp. écriture). Ce temps pire consiste à considérer le cas le plus défavorable (ex. la FIFO du MCM est pleine, la RAM est toujours ouverte sur une mauvaise ligne $t^{\max}_{read} = 16x + t_i + t_p + t_a + 16d$). A noter de plus, que les *refresh* se produisent à n'importe quel moment et ferment toutes les banques. Au pire, un *refresh* s'insère entre deux accès à la même ligne et le coût est de $t_p + t_a + t_{ref}$. On connaît néanmoins le nombre max de *refresh* sur un intervalle donné donc on ajoute un coût fixe. Cette première approche est très pessimiste.

Modélisation en Uppaal

Nous avons donc opté pour une approche alternative : modéliser les comportements possibles comme un réseau d'automates temporisés avec Uppaal. Pour le MPC8641D, nous avons $t_a = 8$, $t_p = 8$, $t_w = 4$, $t_r = 4$, $t_{ref} = 64$, $t_i = 34$, $x = 6$ et $d = 12$. On connaît statiquement grâce au placement les données qui sont écrites (resp. lues) à chaque tranche : on stocke ces informations dans des listes (1 par cœur, 1 par périphérique). Illustrons l'idée sur l'exemple d'une lecture pour les deux cœurs accédant à le même banque. Il y a quatre automates : le premier décrit le comportement d'un cœur (cet automate est instancié deux fois), le deuxième le bus de requête, le troisième la mémoire et son contrôleur, et le dernier le bus de données.

L'automate générique d'un cœur est montré ci-dessus. L'émission des requêtes commencent à une certaine date entre $[0, Dephasage]$ car les démarrages des cœurs en lecture sont asynchrones. Dans l'état *emission_req* : les requêtes sont émises toutes les unités de temps tant que la taille de la FIFO MCM le permet ; le cœur peut également recevoir les données demandées. La fin de la tranche correspond au moment où toutes les requêtes ont été émises via *chan send_addr* ($k + i = NbRequest$) et les données reçues via *chan recv_data* ($p = NbRequest$). Il faut deux boucles pour *send_addr* à cause des problèmes de débordement de tableau en Uppaal. Il y a également deux boucles de remise à zéro de l'horloge quand le MCM est plein ($nbreqmcm == MCM_FIFO_size$) ou lorsque le cœur attend la fin des données.



Le bus de requêtes est montré ci-dessus. Les requêtes sont récupérées des cœurs via *chan send_ addr* puis retransmises au bout de *delay_ req* unités de temps au contrôleur via *chan ask_ page*. Les requêtes sont transmises tant que la FIFO du contrôle *nbdatactrl* n'est pas pleine. Sinon, l'automate entre dans un état d'attente *wait* et tente de retransmettre une requête toutes les unités de temps. Le nombre de requêtes en cours dans le MCM est compté par la variable *nbreqmcm*.

Le contrôleur et la mémoire sont représentés dans l'automate figure 3.8. Initialement l'automate est

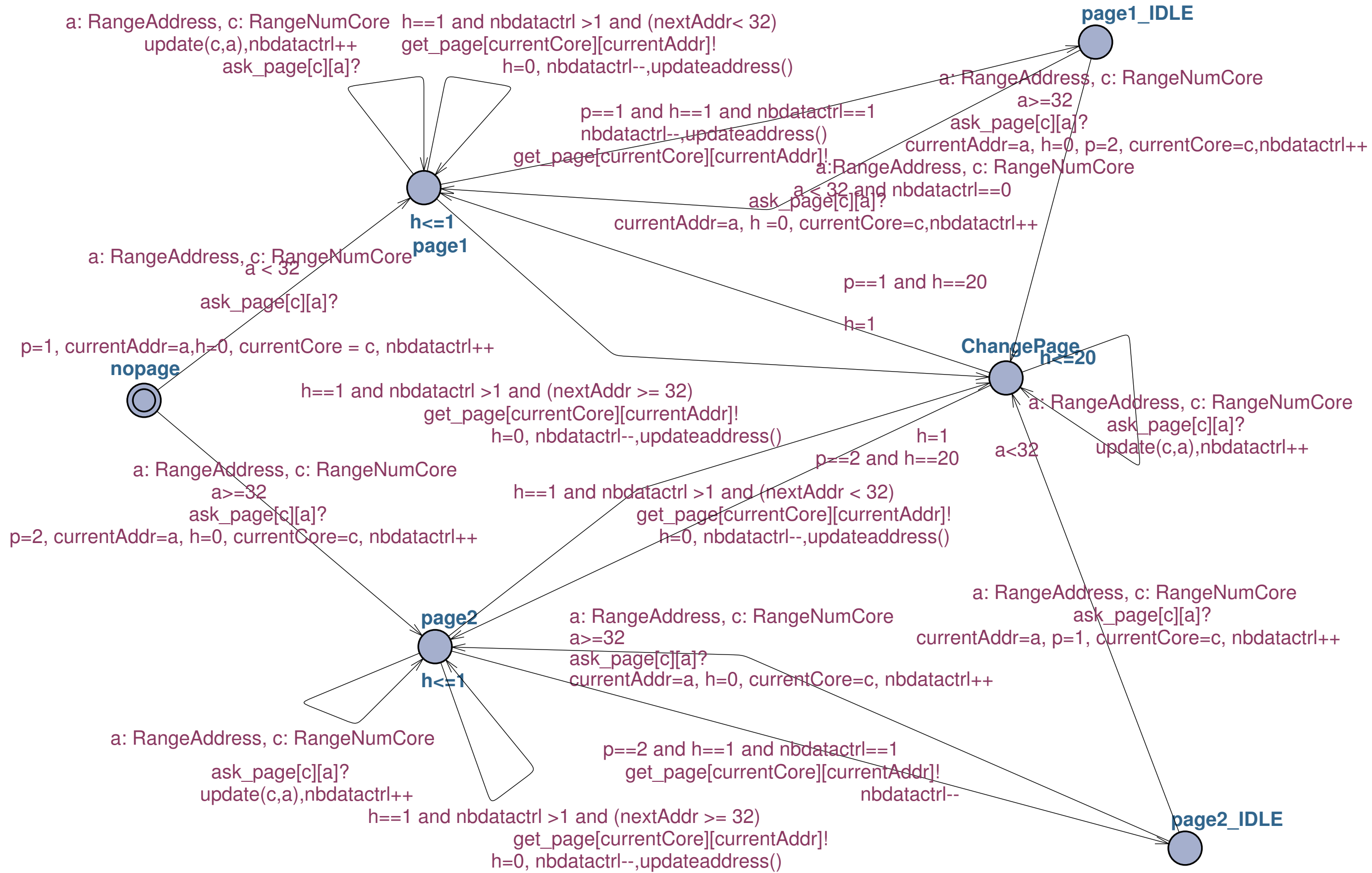
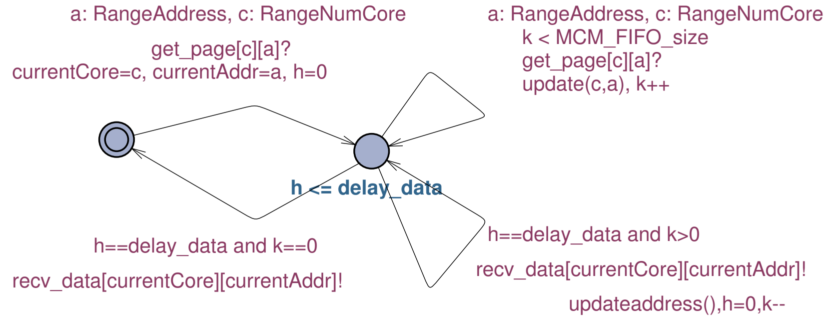


FIGURE 3.8 – Automate temporisé de la mémoire

dans *nopage*. Puis le contrôleur reçoit une requête qui est transmise à la RAM via *chan ask_ page* et on suppose que la page demandée est ouverte. Soit l'adresse est dans la première (*page1*), soit dans la deuxième (*page2*). Les deux comportements sont identiques. Décrivons de ce fait uniquement la partie haute de l'automate. Le contrôleur peut recevoir à n'importe quel moment d'autres requêtes qui sont stockées dans sa FIFO. Au bout d'une unité de temps, la donnée est envoyée sur le bus de données via *chan get_ page*. Il y a alors 3 cas de figure :

1. aucune autre requête ne se trouve dans le contrôleur, la RAM passe dans l'état *page1_ idle*. A savoir que la page 1 est ouverte, en attente de lecture ;
2. la requête suivante à traiter demande une adresse *NextAddr* dans la même page, donc on retourne dans le même état et cette nouvelle requête sera traitée dans 1 unité de temps ;
3. la requête suivante à traiter demande une adresse *NextAddr* dans l'autre page, donc on va dans l'état *ChangePage*.

Une fois dans l'état *ChangePage*, on y reste 20 unités de temps, pour le 'P' et le 'A'.



L'automate du bus de données, montré ci-dessus, décrit la latence pour transmettre la donnée jusqu'aux cœurs et le stockage des données dans le MCM pendant les transmissions en cours. Soit le bus est *idle*, soit il attend *delay_ data* avant de retransmettre via *chan recv_ data*. Il peut également stocker des données en provenance de la RAM.

On utilise ensuite la variable globale *temps_ global* pour calculer le temps maximal pour attendre les états *fin*.

$$A[]temps_global \leq \Delta$$

3.4 Modèle d'exécution 2 : AER

Le deuxième modèle a été présenté dans [DFG14]. L'application avionique n'est plus de type contrôle/commande.

Description du FMS

La figure 3.9 décrit l'architecture fonctionnelle du Flight Management System. Par souci de simplicité, le système est décrit comme un ensemble de groupes fonctionnels mais ce regroupement n'existe pas au niveau implantation.

- Le groupe *sensor* collecte les informations des capteurs tels que le capteur *Anemo-barometric*, le capteur *Pure Inertial Reference System* (IRS) ou le *Global Positioning System* (GPS) ;
- Le groupe *localization* calcule la position probable de l'avion en fonction des données capteurs *Best Computed Position (BCP)* ;
- Le groupe *flight plan* suit la route de vol décidée au décollage et éventuellement modifiée par le pilote ;
- Le groupe *trajectory* utilise la position BCP pour calculer les trajectoires locales (« *profiles* ») à suivre de façon à respecter le plan de vol. Ce calcul implique des modèles physiques et des contraintes complexes telles que respecter l'horaire d'arrivée ou minimiser la consommation de carburant ;
- Le groupe *guidance* traduit les profiles en valeurs d'angle pour le pilote automatique des commandes de vol ;
- Le groupe *nearest* calcule régulièrement la liste des aéroports les plus proches, afin d'aider le pilote en cas d'atterrissage d'urgence ;
- Le groupe *display* ne fait pas partie du FMS mais les données calculées par le FMS doivent y être afficher en temps réel et avec les valeurs correctes.

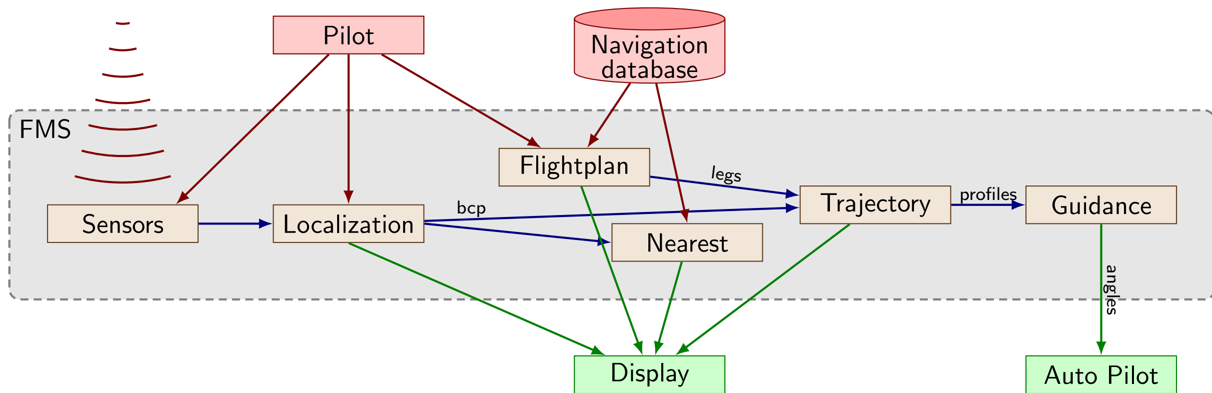


FIGURE 3.9 – Description fonctionnelle du FMS

Description du modèle

L'architecture cible est un multi-cœurs COTS : le TMS320C6678 de Texas Instrument [Tex13] composé de 8 cœurs. L'avantage principal de cette puce, décrite plus en détail dans la partie 4.3.1, est la possibilité de configurer les caches privés en SRAM. La plate-forme devient ainsi une architecture distribuée avec des accès explicites aux bus et mémoires partagés. Du fait des capacités du TMS, on peut réduire les zones de conflits aux échanges de données entre tâches et I/Os. Plus besoin de pré-charger les codes et données internes comme dans le modèle d'exécution 1.

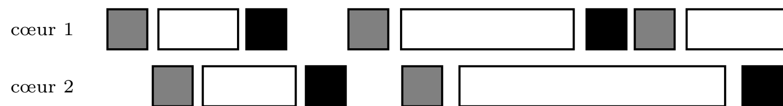


FIGURE 3.10 – Modèle d'exécution AER

Le modèle d'exécution *AER* pour *Acquisition - Exécution - Restitution*, montré dans la 3.10, opère comme suit :

— Les tâches sont découpées en 3 parties :

1. pendant la phase d'acquisition A (blocs gris dans la figure), les données provenant d'autres tâches ou d'entrées sont lues dans un espace partagé puis copiées dans des variables locales ;
2. pendant la phase d'exécution E (blocs blancs dans la figure), la tâche exécute son code fonctionnel. Aucune des variables manipulées n'est utilisée par d'autres tâches ou des périphériques ;
3. pendant la phase de restitution E (blocs noirs dans la figure), la tâche écrit ses sorties dans un espace mémoire partagé ou vers les périphériques.

— Les tâches sont partitionnées sur les cœurs. Les différents types de blocs (A, E ou R) sont séquencés avec un ordonnancement non pré-emptif calculé hors-ligne. Ce placement statique doit vérifier les propriétés suivantes :

1. tous les segments (`.stack`, `.data`, `.text`,...) sont stockés dans les mémoires locales (pour le TMS il s'agit du L2 SRAM). Ainsi, les phases d'exécution n'accèdent pas aux ressources partagées. Leur ordonnancement n'est donc soumis à aucune contrainte.
2. les phases d'acquisition et restitution sont les seules à accéder à des zones partagées. Afin de rester prédictible, elles sont exécutées en isolation d'autres A et R.

La phase de validation est beaucoup plus simple. Il faut toujours appliquer une technique d'analyse de WCET sur les phases d'exécution. Pour les mêmes raisons que celles exposées pour l'analyse du WCET de tranches d'exécution du modèle 1, cela se ramène à l'analyse d'un code séquentiel mono-processeur avec accès à des ressources privées. L'analyse des phases d'acquisition ou de restitution est en revanche beaucoup plus simple. Puisque ces activités sont sans concurrence, il faut déterminer le temps d'accès pire cas en isolation.

3.5 Modèle d'exécution 3 : contentions réduites

Le dernier modèle est une version *affaiblie* des précédents, dans le sens où il accepte des zones temporelles d'interférence, mais il est plus générique car il peut s'appliquer à davantage de cibles. Il résulte de nos travaux avec Woflgang Puffitsch et Eric Noulard, présentés en partie dans [PNP13] et en cours de soumission dans une version journal. Il est défini par les règles suivantes :

Règle 1 : les tâches sont ordonnancées par un séquenceur partitionné non pré-emptif calculé hors-ligne ;

Règle 2 : les codes et données doivent être stockés dans les mémoires locales afin de supprimer les accès implicites aux mémoires partagées. Lorsque l'architecture est basée sur des caches (comme le SCC), cela se traduit par une phase initiale de warm-up et vérification d'absence de caches miss.

Règle 3 : les tâches communiquent par passages de message. Les messages sont stockés dans une zone mémoire spécifique locale dédiée à ce propos (appelé *message passing areas* - MPAs). Nous nous sommes inspirés du concept du MPB de l'Intel SCC ;

Règle 4 : les délais des communications explicites entre cœurs ou avec la mémoire externe ou les périphériques doivent être pré-calculés. Les accès concurrents sont acceptés tant que les délais sont bornés et respectent les échéances. Les bornes sont déterminées par une approche par mesure.

Ce modèle a été défini pour les pluri-cœurs SCC et TILERA, afin de ne pas développer des *flusheur* et *fetcheur*. Il peut également s'appliquer au TMS. Il est également moins contraint que l'AER puisque des écritures et lectures parallèles sont acceptées. Le pire temps de communication est noté t_{comm}. Ce dernier pourrait être calculé par une approche formelle mais du fait de l'absence d'outil d'analyse statique et d'un accès restreint à de la documentation très détaillée des puces, nous nous sommes basés sur des *stressing benchmark* pour déterminer les bornes maximales d'envoi de données et de contention.

Utiliser des *stressing benchmarks* sur les plates-formes est très courant même si le taux de couverture des résultats est difficile à justifier. Les travaux de [NP12] ou [BGG14 ; Bin14] proposent des méthodes génériques et systématiques de recherche de paramètres. Le pseudo-code de notre stressing benchmark, test_run, repose sur les mêmes idées et est montré ci-dessous.

```
void test_run(int do_read, int mode,
              int test_node, int noise_nodes) {
  if (core_id == test_node) {
    long long maximum = 0;
    for (i = 0; i < ITERS; i++) {
      for (k = 0; k < NUM_MPAS; k++) {
        volatile int dummy;
        long long start = read_timestamp();
        if (do_read) {
          dummy = read_mpa(k);
        } else {
          write_mpa(k, dummy);
        }
        long long end = read_timestamp();
        if (i > 0) {
          maximum = max(maximum, end-start);
        }
      }
    }
    record(rdwr, mode, noise_nodes, maximum);
  } else if (core_id < noise_nodes
                 + (test_node < noise_nodes ? 1 : 0)) {
    for (i = 0; i < NOISE_ITERS; i++) {
      volatile int dummy;
      switch (mode) {
        case 0: dummy = read_mpa(test_node); break;
        case 1: write_mpa(test_node, dummy); break;
        case 2: dummy = read_extmem(); break;
        case 3: write_extmem(dummy); break;
      }
    }
  }
}
```

Dans ces expérimentations, le cœur test_node répète des lectures depuis ou des écritures sur les MPAs des autres cœurs et mesure les temps d'accès. Pendant ces mesures, les cœurs noise_nodes chargent le réseau avec des accès à une ressources partagée, par exemple lecture ou écriture sur le MPA du test_node. Les mesures unitaires sont répétées ITERS fois. La fonction test_run est appelée sur chaque cœur et le

nombre de cœurs faisant du bruit varie de 0 à n-1. Les accès en lecture et en écriture sont testés, de même que plusieurs manières de générer du bruit. Les fonctions read_mpa, write_mpa, read_extmem, et write_extmem sont spécifiques à chaque cible et doivent être adaptées.

Pour le SCC, une lecture consiste à faire une requête de lecture et un paquet de réponse. De ce fait, on mesure le round-trip time (RTT) qui donne une borne supérieure de temps de communication. Le RTT sur le SCC est de 10 μs sans contention. Cependant, lorsque 18 cœurs ou plus accèdent en lecture/écriture au MPA du cœur sous test, t_{comm} croît drastiquement. Finalement, nous retiendrons : $t_{comm} = 100$ μs et $n_{cont} = 15$.

Les mêmes tests réalisés sur le TMS320C6678 montrent que les temps en lecture ou écriture sont indépendants du nombre de contention. Cela tient au fait que le nombre de cœurs est faible et que le TERANET a un excellent débit. Pour les différents types de bruit, on observe t_{comm} autour de 130 à 160 ns.

3.6 Conclusion

L'embarquement de plates-formes multi/pluri-cœurs est un enjeu industriel important car non seulement ces processeurs vont devenir les seuls disponibles sur le marché mais également parce qu'ils ouvrent de nouvelles perspectives de conception grâce à leurs capacités de calcul et leur rapport puissance de calcul / consommation électrique. Un des verrous reste le manque de méthodes et outils pour (1) calculer formellement des WCET et (2) assurer le partitionnement entre applications, notamment pour appliquer les principes de *certification incrémentale* (valider chaque application indépendamment les unes des autres).

Nous avons proposé des pistes vraiment prometteuses, basées sur l'utilisation de modèles d'exécution, pour configurer ces plates-formes COTS de sorte qu'un outil d'analyse statique puisse être utilisé ou qu'une argumentation avec les autorités de certification puisse être menée sur la prédictibilité de l'implantation. Nos choix ont à nouveau été guidés par les nombreuses discussions avec nos partenaires industriels et leurs pratiques habituelles d'intégration d'applications avioniques sur les cibles afin de rester le plus proche possible de leur manière de faire.

La prochaine étape consiste à porter ces idées théoriques sur des processeurs réels COTS.

Chapitre 4

Implantations prédictibles sur multi/pluri-cœurs

Dans le chapitre précédent, nous avons présenté les modèles prédictibles théoriques envisagés pour exécuter des programmes PRELUDE sur des cibles multi/pluri-cœurs. Ce chapitre se concentre sur l'implantation des solutions retenues et la mise à disposition au concepteur d'outils pour générer automatiquement des configurations selon les règles retenues dans le modèle d'exécution.

4.1 Contexte

Le développement de logiciels embarqués ainsi que les outils de génération de configuration sont soumis au standard DO 178B/C [RTC08 ; RTC11a]. Cela implique de nombreuses contraintes sur (1) le code embarqué, lequel doit être prédictible, traçable et le plus simple possible ; (2) les services exécutifs, lesquels doivent être prédictibles, observables, minimaux et ne répondant qu'à besoins réels ; (3) les outils de génération, lesquels doivent être qualifiés ou fournis avec des outils de vérification qualifiés. Notre contexte est restreint car nous nous focalisons sur l'intégration et en particulier sur la génération de « *configurations* » de programmes PRELUDE et des nœuds importés associés sur les cibles pourvues d'exécutifs minimalistes.

4.1.1 Besoins et objectifs

L'exécutif bare-metal minimaliste doit offrir trois services élémentaires. Tous les modèles d'exécution considérés imposent des séquenceurs non pré-emptifs et partitionnés, implantés par des dispatcheurs très simples. Le deuxième service attendu est la configuration des mémoires pour stocker de manière temporaire ou permanente les tâches ou les données dans des zones pré-définies. Enfin, le troisième service est la gestion des communications à des moments donnés et selon un paradigme de « passage de message » (excepté modèle 1). Nous avons développé deux exécutifs *bare-metal* : un pour le SCC avec le modèle d'exécution 3 dans le cadre des projets SCC et TOAST déjà mentionnés p 36 ; et un pour le TMS dédié aux modèles 2 et 3 dans le cadre du projet WC$\star$T décrit p 36. La différence entre ces deux modèles sur le TMS consiste uniquement à assurer que les séquenceurs empêchent l'occurrence d'écritures et/ou de lectures en parallèle.

L'outil fourni à l'utilisateur doit calculer une *configuration* (ou un placement) définie par (1) la localité (ou le cœur) exécutant chaque tâche, (2) les zones mémoires hébergeant les sections de manière temporaire ou permanente, (3) les moments de démarrage des tâches. Le *problème de placement* consiste à allouer les tâches sur la plate-forme de sorte que toutes les contraintes fonctionnelles (précédences et échéances) et non fonctionnelles (capacités mémoire et processeur) soient satisfaites. Il s'agit d'une variation du problème NP-complet de sac à dos [GGJY76 ; CGJ96] (*bin packing problem*). Nous utilisons une approche par contraintes pour décrire le problème général de placement de tâche : *étant donné une application multi-périodique, un multi/pluri-cœurs soumis à un modèle d'exécution, comment allouer les tâches sur les cœurs et dans les mémoires afin de respecter les contraintes temporelles et les règles d'utilisation.* Cette approche produit des configurations valides par construction et d'un point de vue certification, le plus simple est alors de qualifier un outil de vérification des résultats obtenus. Des solutions similaires ont

déjà été utilisées et qualifiées en aéronautique. Nos premiers travaux sur le placement ont commencé dans le projet MARTIAC/SHRINK et ont donné lieu à une première publication [BHP08]. L'architecture cible était une architecture distribuée composée de plusieurs processeurs connectés par un bus partagé. Les processeurs étaient soumis à des règles des tranches de communication et exécution similaires à celles du modèle d'exécution 1. L'outillage a ensuite été étendu, toujours dans le projet projet MARTIAC/SHRINK, à une architecture multi-cœurs dans [BCNP12]. Enfin, dans le projet WC⋆T nous avons développé une boîte à outils complète allant du calcul de placement à la génération de fichiers d'inclusion .h pour le TMS. Ainsi, l'utilisateur doit simplement modifier la description de son application ou ajouter des contraintes dans le modèle (ex. forcer une tâche à s'exécuter sur un cœur), et l'outillage regénère les fichiers pour de nouvelles exécutions sans intervention manuelle.

4.1.2 Travaux connexes

Sur la recherche de configurations ou placements hors-lignes

Premières résolutions Une première approche de placement d'ensemble de tâches périodiques sur un multi-processeurs a été proposée dans [XP93 ; Xu93]. L'ordonnancement est non pré-emptif mais la migration est acceptée. Les tâches sont périodiques avec précédence et il est possible d'exprimer des contraintes d'exclusion. A partir d'un séquencement non nécessairement valide calculé par une heuristique, l'algorithme va l'améliorer jusqu'à trouver un séquencement faisable ou alors conclure qu'aucune solution n'existe.

Les auteurs de [PSA97] ont développé un algorithme *branch and bound* pour placer des ensembles de tâches synchrones à échéances sur requête sur une architecture distribuée en tenant compte des temps de communication sur le réseau. Les auteurs de [AS99] ont ensuite étendu ces résultats pour ordonnancer les messages sur le réseau. A partir d'un partitionnement des tâches obtenu par l'algorithme précédent, ils calculent un ordonnancement local et faisable par processeur puis une affectation de priorité pour les messages presque optimale. A nouveau les auteurs utilisent un algorithme *branch and bound*. Le modèle du système est plus général puisque les tâches sont asynchrones à échéances contraintes.

Approches par contraintes Plusieurs travaux ont utilisé des approches par résolution de contraintes pour calculer des ordonnancements hors-ligne, notamment [Eke04]. L'approche proposée par [SW00] se concentre sur la génération d'ordonnancement de systèmes dirigés par le temps communiquant par messages sur un bus partagé. Le partitionnement de tâches est supposé déjà fait. Hladik et al. [HCDJ08] gèrent le placement de tâches sans précédence avec des ordonnancements pré-emptifs à priorités fixes.

Autres approches Il existe d'autres alternatives aux approches par contraintes. Grolleau et al. [GC01] utilisent des réseaux de Petri pour calculer des ordonnancements hors-lignes pour des tâches sur multi-processeurs. Ils acceptent la pré-emption et le migration, tout en essayant de minimiser la migration. [BLR05] utilisent des automates temporisés à coûts ce qui offre une très grand flexibilité mais est sensible à l'explosion combinatoire. Plusieurs approches reposent sur des heuristiques. La méthodologie AAA (*Algorithm Architecture Adequation*) [Sor96] et l'outil associé SYNDEX [GLS99] permettent de concevoir des placements partitionnés non pré-emptifs sur architectures multi-processeurs. Le système est modélisé par des graphes : (1) la partie fonctionnelle est décrite par un DAG (Directed Acyclic Graph) où les nœuds sont les opérations (similaires à des nœuds LUSTRE) et les arêtes les communications entre opérations. (2) tandis que l'architecture matérielle est décrite par un graphe de ressources, les arêtes représentant les liens de communication entre ressources. L'adéquation cherche à optimiser le placement de l'architecture fonctionnelle sur le matériel en ordonnançant et distribuant les opérations et les communications. Le résultat est un ordonnancement non pré-emptif partitionné synchrone.

Sur l'implantation d'exécutifs prédictibles

La littérature porte en général sur le développement de noyaux exécutifs, de RTOS ou d'hyperviseurs. Les codages en bare-metal font moins l'objet de publications académiques. Nous avions réalisé en 2012 un état des lieux en programmation bare-metal sur la page web `http://sites.onera.fr/scc/baremetal`. La plupart des machines COTS achetées à l'ONERA (SCC, TMS, TILERA, KALRAY) offrent un support utilisateur pour coder en bare-metal : séquence de boot, configuration de la mémoire, quelques services de gestion des entrées / sorties et quelques librairies (notamment mathématique).

4.2 Techniques de placement de tâches

Cette partie décrit la formulation générale de recherche de placements valides et les déclinaisons selon les spécificités des différents modèles.

4.2.1 Rappels

Le problème de recherche d'un placement non pré-emptif partitionné est équivalent à celui du sac à dos (ou bin packing) rappelé ci-dessous.

Définition 13 (Enoncé du problème de bin packing). *Soit un ensemble d'articles $X = \{x_1, \ldots, x_n\}$, une fonction taille $w : X \to \mathbb{N}$ qui associe à chaque objet sa taille, un entier C et un entier k. Un* rangement *$\mathcal{R}$ est une application $X \to [1..k]$ telle que :*

1. *$\mathcal{R}(x) = j$ signifie que x est rangé dans le sac j,*
2. *$\forall j \in [1..k]$, $\sum_{x \in X | \mathcal{R}(x)=j} w(x) \leq C$.*

Question : *Existe-t-il un rangement $\mathcal{R}$ des objets de X dans k sacs de capacité C ?*

Une tâche peut être vue comme un objet et un processeur comme un sac. Afin de traiter les différents paramètres des nœuds (WCET, taille du code, taille des données,...), il faut considérer le problème du sac à dos multi-dimensionnel.

Définition 14 (Enoncé du problème de bin packing multi-dimensionnel). *On associe à chaque objet d attributs constituant sa taille. Soit un ensemble d'articles $X = \{x_1, \ldots, x_n\}$, une fonction taille $w : X \to \mathbb{N}^d$ qui associe à chaque objet les attributs de sa taille, un vecteur d'entiers $(C_1, \ldots, C_d)$ indiquant la capacité des sacs selon chaque dimension et un entier k.*

Un rangement *$\mathcal{R}$ est une application $X \to [1..k]$ telle que :*

1. *$\mathcal{R}(x) = j$ signifie que x est rangé dans le sac j,*
2. *$\forall j \in [1..k], l \in [1..d]$, $\sum_{x \in X | \mathcal{R}(x)=j} [w(x)]_l \leq C_l$ où $[w(x)]_l$ est la valeur de w dans la dimension l.*

Question : *Existe-t-il un rangement $\mathcal{R}$ des objets de X dans k sacs de capacité $(C_1, \ldots, C_d)$?*

Méthode retenue : résolution par contraintes

Un problème de satisfaction de contraintes [RvW06] est décrit par un triplet (X, D, C) où :

1. $X = \{x_1, \ldots, x_n\}$ est un ensemble fini de variables ;
2. D est une fonction qui associe à chaque variable x_i son domaine $D(x_i)$, i.e. l'ensemble des valeurs prises par x_i ;
3. $C = \{C_1, C_2, ..., C_k\}$ est l'ensemble de contraintes auxquelles sont soumises les variables.

Une solution au problème est de choisir une valeur pour chaque variable x_i dans $D(x_i)$ satisfaisant toutes les contraintes. Lorsqu'il y a plusieurs solutions, il peut être intéressant d'exprimer des critères de qualité à optimiser.

Exemple 23. *Considérons 4 tâches à placer sur 2 processeurs. Alors, (X, D, C) est donné par :*

1. *$X = \{x_1, x_2, x_3, x_4\}$ représente les 4 tâches ;*
2. *$D(x_1) = D(x_2) = D(x_3) = D(x_4) = \{0, 1\}$ où x_i vaut 0 si x_i est placé sur le premier processeur, et est sur le deuxième sinon ;*
3. *$C = \{x_1 = x_2, x_2 \neq x_4\}$ décrit l'ensemble des contraintes. Ici on spécifie que x_1 et x_2 doivent être sur le même processeur (on parle de collocation). De plus, x_2 et x_4 ne doivent pas être sur le même processeur (on parle d'exclusion).*

Il y a plusieurs solutions à ce problème : $(x_1, x_2, x_3, x_4) \in \{(0, 0,\ 1, 1), (0, 0, 0, 1), (1, 1, 1, 0), (1, 1, 0, 0)\}$. Si on ajoute le critère $f(S) = x_1 + x_3$ alors la seule solution maximisant f est $\{(1, 1, 1, 0)\}$.

4.2.2 Données du problème

La première étape consiste à définir proprement les entrées du problème. Les données d'entrée du problème sont l'application et le modèle de la plate-forme.

Modèle du système

Le modèle d'entrée applicatif est celui de PRELUDE. On repart de la description du modèle de tâches générées décrit dans la partie 2.3.3.

Définition 15 (Modèle du système). *Un système $\mathcal{P}$ est un n-uplet $\langle \mathcal{S}, \mathcal{R}, \mathcal{C}, t_{tick} \rangle$ tel que :*

1. *$\mathcal{S} = \{\tau_1, \ldots, \tau_n\}$ est un ensemble fini de tâches où chaque tâche est un programme séquentiel qui termine tel que :*
   (a) *size_task : $\mathcal{S} \to \mathbb{N}$ associe à chaque tâche τ_i le nombre de lignes mémoire nécessaires pour stocker l'exécutable (le code, les données internes, la pile (*stack*)) de τ_i ;*
   (b) *T : (resp. O : ; D :, C :) $\mathcal{S} \to \mathbb{N}$ donne la période (resp. date d'activation ; échéance relative ; WCET) de la tâche. On suppose que $D(\tau_i) \leq T(f_i)$;*
   (c) *On note par $\tau_{i.j}$ le j-ème job (ou instance) de la tâche τ_i ; chaque job a une date de réveil $r_{i.j}$ et une échéance absolue $d_{i.j}$.*

$$\tau_i = \langle \tau_{i.0}, \tau_{i.1}, \ldots \rangle \qquad avec \qquad \tau_{i.j} = (r_{i.j}, d_{i.j})$$

   On a pour chaque job $\tau_{i.j}$:

$$\begin{aligned} r_{i.j} &= O_i + jT_i \\ d_{i.j} &= O_i + jT_i + D_i \end{aligned}$$

2. *$\mathcal{R}$ est une relation de précédence étendue liant les jobs entre eux.*

$$\mathcal{R} = \{\tau_i \xrightarrow{M_{i,j}, L_{i,j}} \tau_j\}$$

   On suppose que la relation de précédence est causale.

3. *$\mathcal{C} = \{b_1, \ldots, b_k\}$ est un ensemble fini de buffers échangés entre les tâches τ_i tel que :*
   (a) *size_buf : $\mathcal{C} \to \mathbb{N} \times \mathbb{N}$ associe à chaque buffer le nombre de cases et la taille en octets d'un élément ;*
   (b) *src : (resp. dst :) $\mathcal{C} \to \mathcal{S}$ décrit le producteur (resp. consommateur) du buffer.*
   (c) *$\mathcal{I} \subseteq \mathcal{C}$ (resp. $\mathcal{O} \subseteq \mathcal{C}$) est l'ensemble des entrées depuis (resp. sorties vers) l'environnement ;*
   (d) *Il faut également spécifier les patrons de lecture et d'écriture dans les buffers :*

$$\begin{aligned} rdpat &= \langle r_1, \ldots, r_n \rangle, \ r_i \in \{true, false\} \\ wrpat &= \langle w_1, \ldots, w_m \rangle, \ w_i \in \{true, false\} \end{aligned}$$

   C'est une version abstraite des patrons définis p. 20, où le numéro des cases est inutile, car nous placerons le buffer dans son intégralité dans une zone mémoire.

4. *t_{tick} est le top de base en PRELUDE.*

Exemple 24. *Reprenons l'exemple 1 p. 12 dont les tâches ont été explicitées dans l'exemple 4 p. 17 et une partie du protocole de communication dans l'exemple 9 p. 20. On note chaque tâche par $\tau = ((T, C, D, O), size_task)$ (la taille est donnée en koctets) et on a*

$$\mathcal{S} = \left\{ \begin{array}{l} \tau_1 = ((24,1,24,0),2), \tau_2 = ((4,1,4,0),2), \tau_3 = ((24,1,24,12),2), \\ \text{sensor}_1 = ((8,1,8,0),1.5), \text{sensor}_2 = ((20,1,20,0),1.5), \\ \text{sensor}_3 = ((28,1,28,0),1.5), \text{actuator}_1 = ((24,1,24,0),1), \\ \text{actuator}_2 = ((4,1,4,0),1), \text{actuator}_3 = ((24,1,24,12),1) \end{array} \right\}$$

On note chaque buffer par $b = (src, dest, nb_elem, size_elem, rdpat, wrdpat)$ (la taille est donnée en octets) et on a

$$\mathcal{C} = \left\{ \begin{array}{l} b_1 = (\text{sensor}_2, \tau_2, 1, 4, \langle t \rangle, \langle t \rangle), b_2 = (\text{sensor}_3, \tau_2, 1, 4, \langle t \rangle, \langle t \rangle), \\ b_3 = (\tau_2, \text{actuator}_2, 1, 4, \langle t \rangle, \langle t \rangle) b_4 = (\text{sensor}_1, \tau_1, 1, 4, \langle t \rangle, \langle t, f, f \rangle) \\ b_5 = (\tau_1, \text{actuator}_1, 1, 4, \langle t \rangle, \langle t \rangle), b_6 = (\text{actuator}_1, \tau_3, 2, 4, \langle t \rangle, \langle t \rangle) \\ b_7 = (\text{actuator}_2, \tau_3, 2, 4, \langle t \rangle, \langle t, f, f, f, f, f \rangle), b_8 = (\tau_3, \text{actuator}_o_3, 1, 4, \langle t \rangle, \langle t \rangle) \end{array} \right\}$$

Enfin, les contraintes de précédence sont données ci-dessous, à noter que tous les $L_{i,j} = 1$:

$$\mathcal{R} = \left\{ \begin{array}{l} \text{sensor}_{2.0} \to \tau_{2.0}, \text{sensor}_{3.0} \to \tau_{2.0}, \tau_{2.0} \to \text{actuator}_{2.0}, \\ \text{sensor}_{1.0} \to \tau_{1.0}, \tau_{1.0} \to \text{actuator}_{1.0}, \\ \text{actuator}_{2.0} \to \tau_{3.0}, \text{actuator}_{1.0} \to \tau_{3.0}, \tau_{3.0} \to \text{actuator}_{3.0} \end{array} \right\}$$

Enfin, $t_{tick} = 1$ ms.

Modèle de la plate-forme

La plate-forme est constituée des mécanismes matériels et des règles des modèles d'exécution.

Définition 16 (Modèle de la plate-forme). *La plate-forme est composée de :*

- N *cœurs ;*
- size_L2 *est la taille du cache L2 privé ;*
- *spécificités du modèle d'exécution :*
  1. *Pour le modèle d'exécution 1 :*
     - size_L1 *est la taille du cache L1 privé ;*
     - *politique des caches. La fonction* set_addr : $\mathbb{N} \rightarrow \mathbb{N}$ *décrit dans quel ensemble d'un cache une adresse abstraite sera stockée ;*
     - *séquenceur synchrone statique : longueur* length_S, *pour chaque cœur q le nombre de tranches d'exécution T_q et les séquences de (*date_start,length_fetch,length_exec,length_flush*).*
  2. *Pour les modèles 2 et 3 :*
     - size_MPA *est la taille du MPA ;*
     - Δ_{clock} *est la précision de l'horloge ;*
     - t_{comm} *est le pire temps de communication ;*
     - n_{cont} *est le nombre maximal de contentions admissibles par le plate-forme.*

Exemple 25. *Le pluri-cœurs SCC avec le modèle d'exécution 3 est modélisé par : $N = 48$, $size_L2 = 256$, $\Delta_{clock} = 40\mu s$, $size_MPA = 16$, $n_{cont} = 15$, $t_{comm} = 100\ \mu s$.*

Le multi-cœurs Freescale MPC8641D [Fre08] avec le modèle d'exécution 1 et le séquenceur statique de la figure 3.4 sont modélisés par : $N = 2$, $size_L1 = 32$, $size_L2 = 1024$, politique = associatif 8 voies. Le séquenceur est défini sur $length_S = 10ms$, pour chaque cœur, il y $T_1 = 2$ et $T_2 = 2$ séquences, définies par $\langle(0,1,2,1)(4,1,4,1);(0,1,4,1)(6,1,2,1)\rangle$.

4.2.3 Formalisation du problème

Expression du problème

Pour un système donné, l'objectif est de trouver un placement spatial et temporel des tâches sur la plate-forme sous-jacente. L'ordonnancement est toujours de type partitionné non pré-emptif avec exécution dirigée par le temps. Il faut donc produire deux types de sortie :

1. un placement spatial :
   (a) placement de chaque tâche sur un cœur ;
   (b) placement des buffers de communication dans la zone mémoire. Adresses mémoires pour le modèle d'exécution 1 ou MPA pour les autres approches.
2. un ordonnancement hors-ligne valide. On produit un séquencement des tâches localement sur chaque cœur et également un séquencement des communications entre tâches.

Puisque les tâches ont des dates de réveil variables, nous nous retrouvons confronter au problème de la fenêtre de faisabilité discuté dans la partie 3.2.2 p. 41. Le choix fait pour contourner ce problème est de réduire la fenêtre à :

$$O_{max} + H$$

Ainsi, seul l'ensemble fini de jobs tels que $r_{i.j} < O_{max} + H$ est à traiter. Cette restriction entraîne une résolution sous-optimale et il serait possible de rallonger la fenêtre pour tendre vers l'optimalité au prix d'une complexité plus grande, mais dans les études de cas réalisées, le choix fait était suffisant pour trouver des solutions comme nous le verrons dans la partie 4.2.5. Dans la suite, on note n_i le nombre de jobs de la tâche τ_i dans la fenêtre.

Variables pour les modèles d'exécution 2 et 3

Les variables nécessaires pour résoudre le problème de placement sont les suivantes :

- $p_i \in \{1, \ldots, N\}$ la variable qui associe le numéro du processeur où s'exécute la tâche τ_i ;
- $s_{i.j}$ la date de réveil du job $\tau_{i.j}$;
- $m_i \in \{1, \ldots, N\}$ la variable qui associe le numéro du processeur où est stocké le buffer b_i ;

Les variables du modèle 1 sont légèrement différentes et traitées en détail partie 4.2.6.

4.2.4 Contraintes pour les modèles d'exécution 2 et 3

Contraintes spatiales

La taille des tâches allouées sur un cœur doit être inférieure à la mémoire locale :

$$\forall q \leq N, \sum_{1 \leq i \leq |\mathcal{S}|} (p_i = q) \times size_task(\tau_i) \leq size_L2 \tag{4.1}$$

Les buffers sont placés dans les MPA et ne doivent pas excéder la taille de celui-ci :

$$\forall q \leq N, \sum_{1 \leq i \leq |\mathcal{C}|} (m_i = q) \times size_buf(i).0 \times size_buf(i).1 \leq size_MPA \tag{4.2}$$

Notons que $size_buf(i)$ est une paire ($size_buf(i).0 =$ taille du buffer, $size_buf(i).1 =$ taille de chaque case).

Il faut faire attention pour le SCC car le MPA est partagé par tuile entre deux processeurs. L'équation doit donc être modifiée de la sorte :

$$\begin{aligned} &\forall q \leq N, \ tel\ que\ q = 0 \mod 2, \\ &\textstyle\sum_{1 \leq i \leq |\mathcal{C}|} (m_i = q \vee m_i = q+1) \times size_buf(i).0 \times size_buf(i).1 \leq size_MPA \end{aligned} \tag{4.3}$$

Contraintes temporelles

Chaque job doit respecter son échéance et sa date de réveil :

$$\forall i \leq |\mathcal{S}|, \forall k \leq n_i, \ s_{i.k} + C_i \leq d_{i.k} \wedge s_{i.k} \geq r_{i.k} \tag{4.4}$$

Deux tâches ne peuvent accéder en même temps au même processeur. Ainsi, soit le job de la première tâche se fait avant ou après n'importe quelle instance de la deuxième :

$$\forall i, j \leq |\mathcal{S}|, i \neq j, \forall k \leq n_i, l \leq n_j, \ p_i = p_j \implies s_{i.k} + C_i \leq s_{j.l} \vee s_{j.l} + C_j \leq s_{i.k} \tag{4.5}$$

Il faut respecter les contraintes de précédence ;

$$\forall i, j \leq |\mathcal{S}|, i \neq j, \forall k \leq n_i, l \leq n_j, \ \tau_{i.k} \to \tau_{j.l} \implies s_{i.k} + C_i \leq s_{j.l} \tag{4.6}$$

Afin de gérer les comportements autour de la fenêtre, on considère l'ensemble des jobs réveillés autour de la fenêtre :

$$\mathcal{H} := \{(\tau_{i.k}, \tau_{i.l}) \mid O_{max} \in]r_{i.k}; d_{i.k}[\wedge r_{i.k} + H = r_{i.l}\} \tag{4.7}$$

Pour assurer que l'ordonnancement est consistent, on impose les contraintes suivantes :

$$\forall (\tau_{i.k}, \tau_{i.l}) \in \mathcal{H}, s_{i.k} + C_i \leq O_{max} \Rightarrow s_{i.l} + C_i \leq O_{max} + H \tag{4.8}$$

$$\forall (\tau_{i.k}, \tau_{i.l}) \in \mathcal{H}, s_{i.k} + C_i > O_{max} \Rightarrow s_{i.l} = s_{i.k} + H \tag{4.9}$$

Il faut également prendre en compte la précision d'horloge Δ_{clock} pour éviter un recouvrement d'exécution de jobs en précédence sur des cœurs différents.

$$\forall i, j \leq |\mathcal{S}|, i \neq j, \forall k \leq n_i, l \leq n_j, \ (\tau_{i.k} \to \tau_{j.l} \wedge p_i \neq p_j) \implies s_{i.k} + C_i + \left\lceil \frac{\Delta_{clock}}{t_{tick}} \right\rceil \leq s_{j.l} \tag{4.10}$$

Cette contrainte est davantage expliquée dans la partie implantation bare-metal p. 65.

Particularités du modèle d'exécution 2

Il faut prendre en compte la contrainte d'exclusion des acquisitions et des restitutions. Dans le modèle Prelude, une acquisition est un *sensor* tandis qu'une restitution est un *actuator* :

$$\forall i, j \leq |\mathcal{S}|, i \neq j, \forall k \leq n_i, l \leq n_j, \tau_i, \tau_j \in \{actuator, sensor\} \implies s_{i.k} + C_i \leq s_{j.l} \vee s_{j.l} + C_j \leq s_{i.k} \tag{4.11}$$

Exemple 26. *Reprenons le système de l'exemple 24 à allouer sur le* Tms*. Le solveur place toutes les tâches sur le premier cœur, en effet l'application est toute petite.*

Particularités du modèle d'exécution 3

Il faut rajouter les temps de communication : on transforme l'équation 4.10, en

$$\forall i, j \leq |\mathcal{S}|, i \neq j, \forall k \leq n_i, l \leq n_j, (\tau_{i.k} \rightarrow \tau_{j.l} \wedge p_i \neq p_j) \implies s_{i.k} + C_i + \left\lceil \frac{\Delta_{clock} + t_{comm}}{t_{tick}} \right\rceil \leq s_{j.l} \quad (4.12)$$

4.2.5 Evaluation

Un prototype écrit en OCAML traduit automatiquement des programmes PRELUDE dans un format compréhensible par le solveur de contraintes d'ILOG OPL [IBM14].

Name	Tasks	Jobs	Precs.	Buffers	O_{max}	H	Utilization
FAS	19	629	548	26	500	10000	1.696
FAS_c	236	5907	6794	332	1000	10000	10.340
ROSACE	14	61	44	20	0	200	1.595
Asm	375	2440	2480	514	0	2400	5.063
CDV_m	311	311	49	1838	0	10000	21.444
CDV	511	13276	5017	980	0	400000	2.421
Master	399	1825	1446	0	0	80000	0.746

TABLE 4.1 – Propriétés des ensembles de tâches

La table 4.1 donne quelques cas d'étude. La première application est le système de guidage (*Flight Application Software* - FAS) de l'ATV (Automated Transfer Vehicle), le véhicule spatial conçu par EADS Astrium Space Transportation pour atteindre la station internationale (International Space Station). Cette spécification vient de la thèse de Julien Forget [For09], la version simplifiée est l'exemple du manuscrit et la version complète est partiellement décrite en annexe. Asm, CDV et CDV_m sont des exemples de commandes de vol électriques. La particularité de CDV est d'être mono-périodique. Master dérive d'une application avionique. Tous les attributs (périodes, échéances, WCETs) sont donnés en millisecondes. Nous avons également évalué la performance de notre approche sur des ensembles de tâches générés automatiquement. Ces ensembles contiennent des tâches de périodes 100, 200, 300, 400, ou 600 unités de temps avec des dates de réveil jusqu'à 600. De ce fait, $O_{max} \leq 600$ et $H = 6000$ pour ces ensembles. Les WCETs sont générés de façon à atteindre une utilisation de 32 en suivant la méthodologie proposée par [BB04]. Générer des précédences est plus complexe donc la plupart des ensembles générés n'ont ni précédence ni communication (contrairement aux études de cas de la table précédente).

La table 4.2 montre les résultats de notre évaluation sur (1) les cas d'étude industriels, (2) le modèle d'exécution 3 et (3) les cibles SCC, TMS et TILERA. La colonne « Prep. » donne le temps pour préprocesser l'ensemble de tâches. Les colonnes « Vars. » et « Constrs. » montrent le nombre de variables et de contraintes reportées par OPL, tandis que la colonne « Memory » indique la taille de la mémoire utilisée. Les colonnes « Solve » et « Br. » fournissent le temps nécessaire à la résolution des contraintes et le nombre de branches explorées par le solveur pour trouver une solution. La plupart des exemples finissent en moins de 6 min.

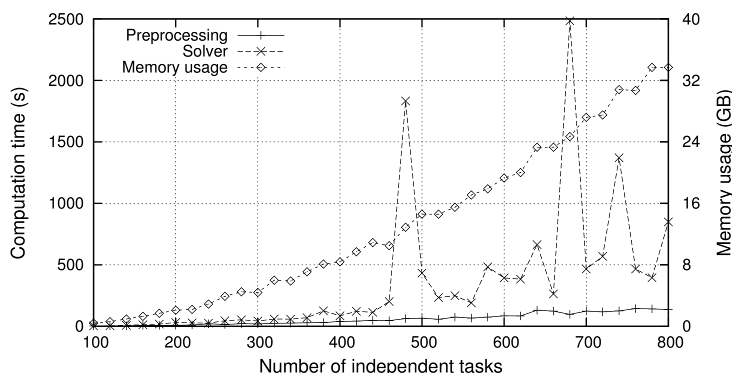


FIGURE 4.1 – Temps de calcul et utilisation mémoire pour les tâches générées

Test Case		Prep. (s)	Vars.	Constrs.	Memory	Solve (s)	Br.
FAS							
	SCC	0.18	648	167846	209.7 MB	1.73	1
	TMS	0.20	648	23614	30.8 MB	0.17	1
	TILERA	0.17	648	186182	225.8 MB	1.95	1
FAS_c							
	SCC	16.06	6143	6205115	6.8 GB	114.81	9003
	TMS			utilisation > 8			
	TILERA	15.60	6143	6439631	7.1 GB	108.48	6972
ROSACE							
	SCC	0.01	75	119718	136.8 MB	1.19	1
	TMS	0.01	75	8822	11.8 MB	0.07	1
	TILERA	0.01	75	133842	151.0 MB	1.34	1
Asm							
	SCC	7.83	2815	5586239	6.4 GB	252.07	13624
	TMS	7.63	2815	2736759	3.1 GB	198.25	30484
	TILERA	7.72	2815	5948975	6.8 GB	212.87	9433
CDV_m							
	SCC	0.43	622	10913509	13.5 GB	323.08	1359
	TMS			utilisation > 8			
	TILERA	0.43	622	12247717	14.7 GB	320.34	650
CDV							
	SCC	55.74	13787	40542879	45.3 GB	1117.45	14393
	TMS	54.23	13787	23300519	25.2 GB	679.63	14706
	TILERA	54.54	13787	42829539	47.4 GB	1302.77	14777
Master							
	SCC	5.16	2224	2712032	3.2 GB	116.81	2225
	TMS	5.27	2224	2664056	3.1 GB	113.81	2225
	TILERA	5.23	2224	2697656	3.2 GB	114.92	2225

TABLE 4.2 – Résultats du placement pour les études de cas

La figure 4.1 montre les résultats de l'évaluation pour les ensembles de tâches générés en terme de temps de calcul et d'utilisation mémoire pour SCC. A nouveau, « Preprocessing » se réfère au temps requis pour pré-processer l'ensemble de tâches, tandis que « Solver » représente celui d'OPL pour trouver la solution. Pour tous les cas, OPL trouve une solution en moins de 2500 secondes avec une utilisation mémoire maximale de 34 GB.

4.2.6 Cas du modèle d'exécution 1

Expression du problème

Etant données une application conforme à la définition 15 et une architecture tranchée, l'objectif est de trouver un placement des tâches et des données sur la cible. Le résultat définit statiquement les adresses mémoire abstraites des instructions et des données, les tranches où les tâches s'exécutent, les tranches où les I/O sont émis et le motif de répétition de placement.

Variables

Les variables du modèle d'exécution 1 sont un peu différentes de celles des autres approches, on note :
- $p_{i.j} \in \{1, \ldots, N\}$ la variable qui associe le numéro du processeur où s'exécute l'instance $\tau_{i.j}$. L'ordonnancement n'est pas partitionné au sens strict, mais on peut voir le système comme l'ensemble des jobs déroulés sur la longueur du séquencement ;
- $s_{i.j}$ la variable qui associe le numéro de la tranche dans laquelle s'exécute le job $\tau_{i.j}$;

— $a_{i.j} \in \mathbb{N}$ associe à chaque tâche τ_i et numéro de ligne $j \in size_task(\tau_i))$ une adresse mémoire abstraite ;
— $m_{i.j} \in \{1, \ldots, N\}$ la variable qui associe une adresse mémoire abstraite à chaque buffer b_i et numéro de ligne $j \in size_buf(i).0 \times size_buf(i).1)$ une adresse mémoire abstraite.

Contraintes

La séquence est hors-ligne est définie sur $H' = ppcm(H, length_S)$ où H est l'hyper-période des tâches $H = ppcm(T_i)$. On note dans la suite, pour tout $q \leq N$, $T'_q = \frac{H'}{length_S} \cdot T_q$ pour dérouler les tranches jusqu'à la longueur de la séquence.

Contraintes spatiales Les contraintes du problème sont architectures-dépendantes puisque les caractéristiques matérielles sont codées en dur. Les instructions et les variables, allouées dans chaque tranche d'exécution, doivent tenir dans le cache L2 et les données dans le cache L1D. L'équation 4.1 est modifiée pour étudier chaque tranche :

$$\forall q \leq N, \forall t \leq T'_q, \\ \sum_{1 \leq i \leq |\mathcal{S}|} (p_{i.j} = q) \times (s_{i.j} = t) \times size_task(\tau_i) \leq size_L2$$

et pour les données, il faut regarder si le consommateur ou le producteur est dans la tranche (mais attention à ne pas compter deux fois le buffer si les deux sont présents) :

$$\forall q \leq N, \forall t \leq T'_q, \\ \sum_{1 \leq i \leq |\mathcal{C}|} \big(((p_{src(i).j} = q) \times (s_{src(i).j} = t)) \vee ((p_{dest(i).j} = q) \times (s_{dest(i).j} = t))\big) \\ \times size_buf(i).0 \times size_buf(i).1 \leq size_L1$$

Contraintes temporelles Chaque job doit respecter ses échéances :

$$\forall i \leq |\mathcal{S}|, \forall k \leq n_i, \\ date_start(s_{i.k}, p_{i.k}) + length_fetch(s_{i.k}, p_{i.k}) \geq r_{i.k} \wedge \\ date_start(s_{i.k}, p_{i.k}) + length_fetch(s_{i.k}, p_{i.k}) + length_exec(s_{i.k}, p_{i.k}) \leq d_{i.k}$$

L'ensemble des tâches allouées dans une même tranche doit avoir assez de temps CPU :

$$\forall q \leq N, \forall t \leq T'_q, \sum_{1 \leq i \leq |\mathcal{S}|} (p_{i.j} = q) \times (s_{i.j} = t) \times C_i \leq length_exec(t, p)$$

Il faut respecter les contraintes de précédence ;

$$\forall i, j \leq |\mathcal{S}|, i \neq j, \forall k \leq n_i, l \leq n_j, \\ \tau_{i.k} \to \tau_{j.l} \wedge (s_{i.k} \neq s_{j.l} \vee p_{i.k} \neq p_{j.l}) \implies \\ date_start(s_{i.k}, p_{i.k}) + length_fetch(s_{i.k}, p_{i.k}) + length_exec(s_{i.k}, p_{i.k}) + length_flush(s_{i.k}, p_{i.k}) \\ \leq date_start(s_{j.l}, p_{j.l})$$

Contraintes adresses mémoires Il faut également tenir compte des règles d'associativité des caches. Prenons l'exemple d'un cache 8-voies, pour lequel pas plus de 8 lignes doivent être stockées dans le même ensemble, i.e. $set_addr : \mathbb{N} \to [1, 8]$. Cela qui s'exprime par :

$$\forall c \in [1, 8], \forall q \leq N, \forall t \leq T'_q, (\sum_{i,k} (p_{i.k} = q) \times (s_{i.k} = t) \times \sum_{l} (set_addr(a_{i.l}) = c)) \leq 8 \tag{4.13}$$

Les buffers doivent être chargés avec chaque producteur et consommateur et doivent tenir dans le L1.

$$\forall c \in [1, 8], \forall q \leq N, \forall t \leq T'_q, \\ (\sum_{1 \leq i \leq |\mathcal{C}|} \big(((p_{src(i).j} = q) \times (s_{src(i).j} = t)) \vee ((p_{dest(i).j} = q) \times (s_{dest(i).j} = t))\big) \\ \times \sum_{l} (set_addr(m_{i.l}) = c)) \leq 8 \tag{4.14}$$

Exemple 27. *Considérons à nouveau l'exemple 24, le séquenceur synchrone statique de la figure 3.5, il n'existe pas de solution : en effet, pour exécuter la séquence i_2, i_3, τ_2, o_2 il faut 4ms auxquelles s'ajoutent les temps de pré-chargement et d'écriture, soit 6ms de bout en bout. Modifions l'ensemble de tâches ainsi :*

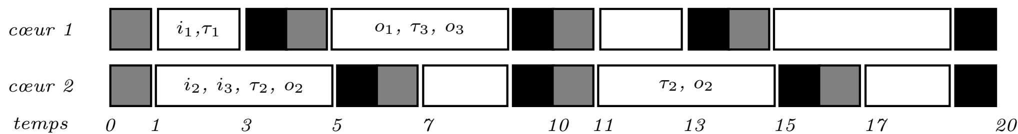
$$\mathcal{S} = \left\{ \begin{array}{l} \tau_1 = ((20,1,20,0),2), \tau_2 = ((10,1,10,0),2), \tau_3 = ((20,1,20,10),2), \\ \text{sensor}_1 = ((10,1,10,0),1.5), \text{sensor}_2 = ((20,1,20,0),1.5), \\ \text{sensor}_3 = ((20,1,20,0),1.5), \text{actuator}_1 = ((20,1,20,0),1), \\ \text{actuator}_2 = ((10,1,10,0),1), \text{actuator}_3 = ((20,1,20,10),1) \end{array} \right\}$$

On suppose que sur la cible : une ligne cache fait 64 octets, le cache est associatif 8 voies, size_L1 $= 32$, *size_L2* $= 1024$. *La fonction* set_addr *est définie par* mod 128. *Alors un code tranché est donné par :*

— *les adresses abstraites pour τ_1 sont* $\{1, \dots, 32\}$, *pour τ_2 sont* $\{33, \dots, 65\}$, *pour τ_3 sont* $\{66, \dots, 98\}$, *pour* sensor$_1$ *sont* $\{99, \dots, 115\}$, *pour* sensor$_2$ *sont* $\{116, \dots, 132\}$, *pour* sensor$_3$ *sont* $\{133, \dots, 149\}$, *pour* actuator$_1$ *sont* $\{150, \dots, 166\}$, *pour* actuator$_2$ *sont* $\{167, \dots, 183\}$, *pour* actuator$_3$ *sont* $\{184, \dots, 200\}$. *Il y a donc au plus 2 éléments par voie et donc aucun conflit.*

— *étant donné la taille des buffers, on peut les grouper dans une même ligne. Ainsi, b_1, b_2 et b_3 sont dans la même ligne* 201, *b_4 et b_5 dans* 202, *b_6, b_7 et b_8 dans* 203.

— *enfin la séquence est définie sur* ppcm$(T_i$, length_S$)$, *soit 20ms dans notre exemple. On donne une exécution valide dans la figure ci-dessous.*



Pour le modèle 1, les solveurs que nous utilisons habituellement n'arrivaient pas à traiter le problème dont la taille était trop grande et nous avons développé un algorithme glouton en C.

4.3 Implantation bare-metal sur multi/pluri-cœurs

Pour implanter les modèles d'exécution et répondre aux besoins de prédictibilité, nous avons développé des exécutifs « *bare-metal* » permettant l'exécution de programmes Prelude et plus généralement de tâches périodiques dépendantes communicantes $\langle \mathcal{S}, \mathcal{R}, \mathcal{C} \rangle$. Il s'agit donc du même jeu d'entrée que SchedMCore figure 3.1. Le programmation bare-metal consiste à développer avec des primitives bas niveau afin d'éviter l'utilisation d'un système d'exploitation et ainsi obtenir des performances optimales, des comportements temps réel complètement maîtrisés, le tout avec des empreintes mémoires minimales. Le désavantage principal est la dépendance directe avec la cible et le coût de re-développement lorsqu'on en change. A partir des différentes librairies développées à l'ONERA, nous avons néanmoins des approches systématiques pour la gestion temporelles et des mémoires. De plus, toujours à des fins de prédictibilité, nous appliquons les règles de codage MISRA [Con08] ainsi que celles recommandées par le consortium MERASA [BBB10].

4.3.1 Cibles multi/pluri-cœurs considérées

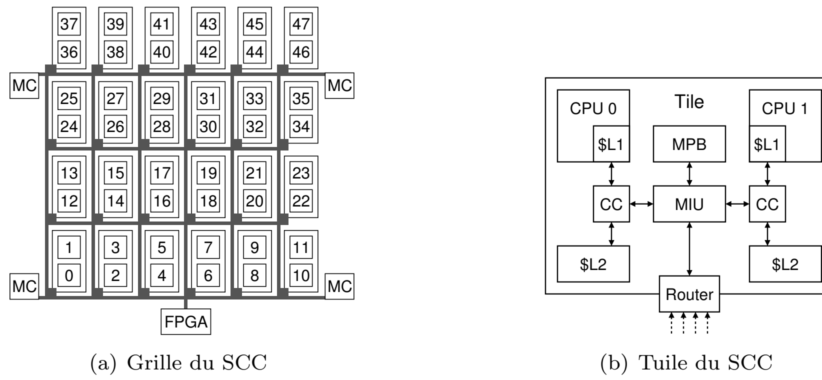
Intel SCC

Le pluri-cœurs Single-chip Cloud Computer (SCC) est une plate-forme de recherche développée par Intel [Int10 ; Int12] à des fins de recherche. Le DTIM était membre de la communauté MARC [1] (Many-core Applications Research Community) et avait accès à un SCC hébergé chez Intel. Wolfgang Puffitsch a porté l'environnement SchedMCore (ordonnancements partitionnés non pré-emptifs uniquement) en bare-metal sur le SCC. Ce travail a été présenté dans [PNP13].

La puce contient 24 tuiles (*tiles*) de dual-core organisées en une matrice 6×4 et connectées via un réseau sur puce NoC (Network on Chip). La structure en grille du SCC est montrée dans la figure 4.2(a), de même que les connexions aux quatre contrôleurs mémoire et avec le monde extérieur.

Chaque tuile (figure 4.2(b)) contient 2 cœurs, un routeur et de la mémoire locale pour permettre du passage de message. Les cœurs sont des Pentium relativement simples contenant des caches L1I de 16KB,

1. `https://communities.intel.com/community/marc`



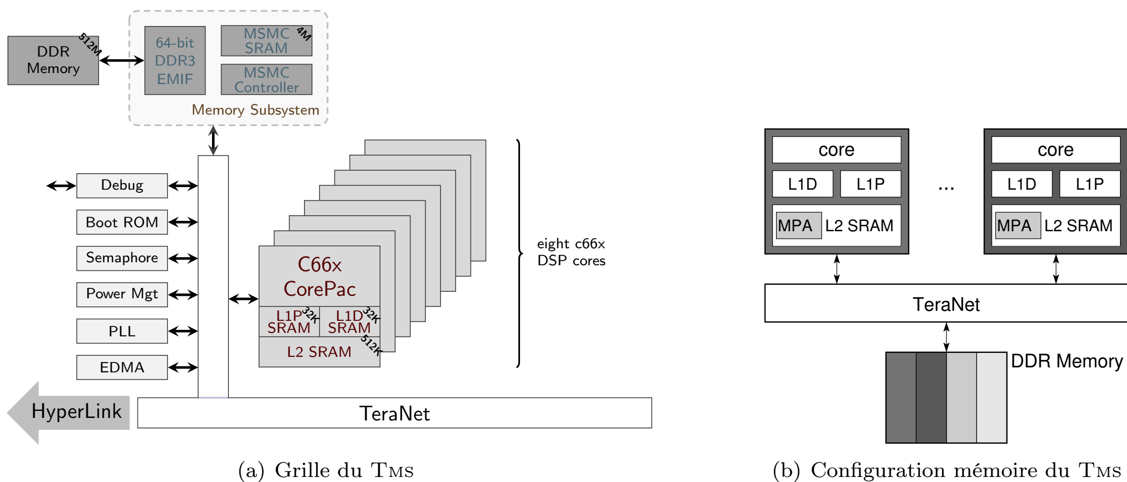
(a) Grille du SCC (b) Tuile du SCC

FIGURE 4.2 – Description du pluri-cœurs Intel SCC

L1D de 16KB et L2 de 256KB. Les cœurs vont à des fréquences entre 100 et 800 MHz tandis que le réseau est rythmé à 800 ou 1600 MHz. Le politique de routage est en XY, c'est-à-dire que les paquets traversent d'abord le long de l'axe X puis Y, et suit une stratégie *wormhole* [Moh98]. La figure 4.2(b) montre les connexions entre cœurs, caches, mesh interface unit (MIU) et routeurs d'une même tuile. Le communication entre cœurs peut se faire de deux façons : à travers la mémoire partagée (RAM) ou par passage de message. La zone mémoire pour envoyer des messages s'appelle le *message passing buffer* (MPB) et fait une taille de 16 KB par tuile.

TMS320C6678

Le multi-cœurs TMS320C6678 [Tex13] est un processeur COTS développé par Texas Instrument. Cette plate-forme a été choisie par Thales dans un projet commun comme support de démonstration à l'exécution prédictible du FMS - présenté dans la partie **??** - sur multi-cœurs. Nous avons porté un environnement bare-metal sur le TMS permettant l'exécution de séquencements partitionnés non pré-emptifs calculés hors-lignes. Ce travail a été présenté dans [DFG14].



(a) Grille du TMS (b) Configuration mémoire du TMS

FIGURE 4.3 – Description du multi-cœurs TMS

Le TMS, montré dans la figure 4.3(a), est un multi-cœurs composé de 8 processeurs DSP TMS320C66x cadencés à 1 GHz. Chaque cœur est basé sur une architecture VLIW et peut exécuter jusqu'à 8 instructions dans un même cycle d'horloge. Les cœurs sont connectés par le réseau sur puce TERANET, qui donne également accès aux modules hardware auxiliaires (ex. interfaces I/O et DMA). Chaque cœur contient plusieurs niveaux de cache L1I de 32 KB, L1D de 32 KB et L2 de 512 KB. Un troisième niveau de cache, un L3 de 4 MB de SRAM (Static Random Access Memory), est partagé par les cœurs.

Le point fort du TMS est sa flexibilité dans la configuration mémoire. En effet, les mémoires L1P, L1D et L2 peuvent être configurées en cache, SRAM ou un mélange des deux. L'utilisation de SRAM pour

le temps réel est un atout, en effet il n'y a pas de *refresh* non prédictible et les temps d'accès sont très faibles. Sur le TMS, lorsqu'une partie des caches est configurée en SRAM, les autres cœurs peuvent y accéder directement grâce à un adressage global. Chaque cœur peut ainsi accéder à ses caches par deux adressages : soit local (uniquement par le cœur), soit global (par tous les cœurs). Par exemple, le cœur 3 peut accéder à son L2 SRAM par l'adresse locale `0x00800000` ou l'adresse globale `0x13800000`. Lorsqu'un cœur accède au L2 SRAM d'un autre cœur, la transaction passe par le TERANET. La configuration de la mémoire est celle décrite dans la figure 4.3(b) : tous les caches L2 sont configurés en SRAM, la partie MPA (message passing Area) est réservée à des opérations précises (gestion temporelle et passage de message).

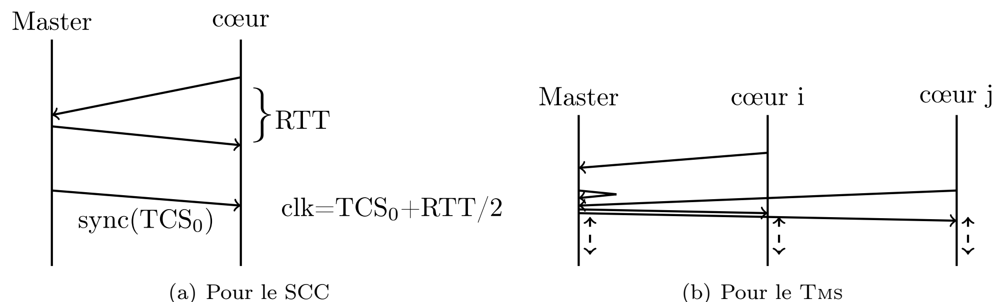
4.3.2 Gestion temporelle

Nous avons rencontré un problème synchronisation sur le SCC, le TMS et le TILERA (qui a servi pour le portage de l'étude de cas ROSACE section 2.4 selon le modèle d'exécution 3). En effet, leurs horloges locales sont synchrones mais les cœurs ne démarrent pas au même moment, ce qui implique qu'elles ont des dates de réveil non prédictibles. De plus, l'accès à l'horloge globale (si elle existe) a un coût élevé et un délai d'accès variable. Le premier besoin est donc d'offrir des services de gestion des horloges locales de façon à avoir une vision temporelle cohérente sur tous les cœurs.

Barrières de synchronisation

Le premier travail est la mise en œuvre d'algorithmes de *synchronisation* de façon à réduire ou maîtriser les dates de réveil des cœurs. Pour chacune des cibles, nous avons défini des *barrières de synchronisation*.

Pour le SCC, l'algorithme est présenté dans [dBNP11] et est illustré dans la figure 4.4(a). Les horloges locales du SCC s'appellent TSC [Int10, §7.1]. Un cœur particulier (0 par exemple) est le *maître*, chaque cœur calcule le Round-Trip Time (RTT) avec le maître, puis reçoit un message du cœur 0 avec son horloge. En ajoutant la moitié du RTT, le cœur a une vision cohérente avec celle du maître. Cette barrière *synchronise à peu près les cœurs* avec une précision de 4 μs.



(a) Pour le SCC (b) Pour le TMS

FIGURE 4.4 – Algorithmes de synchronisation

Pour le TMS, l'algorithme est décrit dans [DFG14]. L'horloge locale est obtenue en lisant 2 registres : *TCSL* et *TCSH*. Ces registres sont synchrones mais avec des dates de réveil non prédictibles entre cœurs. La barrière est basée sur l'écriture de flags dans une zone spécifique du MPA (Message Passing Area) du L2 SRAM. Ainsi, une partie du MPA est réservée aux gestions temporelles sur chaque cœur et l'autre partie est prévue pour l'échange de données. L'algorithme, montré figure 4.4(b), fonctionne comme suit :

- N variables booléennes $\{b_1, \ldots, b_N\}$ sont stockées dans le MPA du cœur 0, qui servira de *master* ;
- 1 variable booléenne c_i est stockée dans le MPA de chaque cœur i ;
- quand le cœur i se réveille, il accède au N variables du cœur 0 et les met à faux. Ensuite, le cœur fait une attente active : tant que sa variable locale c_i est à faux, il écrit vrai dans b_i sur le cœur 0 ;
- quand le cœur 0 se réveille, il met toutes les variables b_i à faux puis fait une attente active sur la condition $\wedge b_i = true$, qui signifie que tous les cœurs sont réveillés. Une fois cette condition vérifiée, il met toutes les variables c_i à vrai ;
- quand le cœur i détecte que sa variable locale c_i est à vrai, il attend 1s avec son horloge locale. Puis, il lit l'heure courante qui devient son offset local ;
- ainsi l'horloge globale partagée = temps local - offset local.

La précision de cette barrière est de 70 cycles, soit 70 ns.

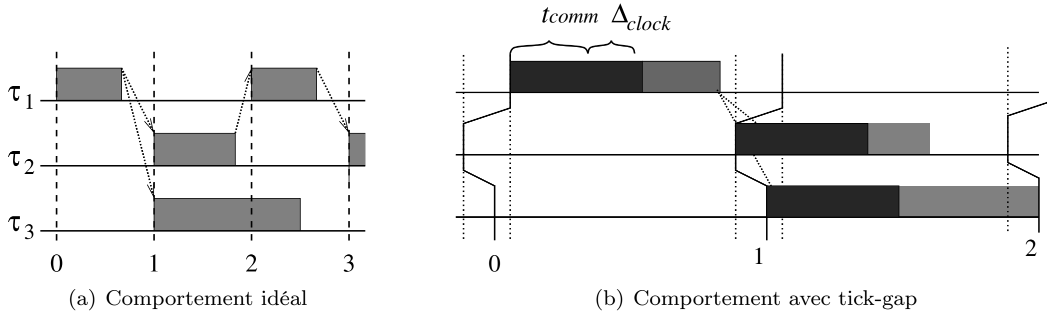
Ordonnancements dirigés par le temps

Le codage des ordonnancements locaux reprend les idées de SchedMCore Runner et repose sur une approche dirigée par le temps, c'est-à-dire que les actions/décisions sont déclenchées par des tops d'horloge. La différence par rapport au codage partie 3.2.1 est que nous n'implantons que des ordonnanceurs non pré-emptifs qu'il est inutile de réveiller tous les tops d'horloge. On connaît le WCET de chaque tâche, ainsi lorsqu'une tâche τ commence au top i, elle s'exécute et quand elle termine, l'exécutif fait une attente active jusqu'au top $i + C_\tau$ ou à un top ultérieur j si aucune action n'est requise entre $i + C_\tau$ et j.

La figure 4.5(a) montre un exemple de décisions prises à des tops d'horloge pour trois tâches τ_1, τ_2 et τ_3 dont les propriétés temporelles sont données dans la table 4.3. Au premier top 0, toutes les tâches vérifient leurs contraintes de précédence : seule τ_1 peut démarrer et les deux autres attendent. τ_1 termine son exécution entre le top 0 et 1. Les autres tâches peuvent donc démarrer au top 1. τ_1 peut commencer sa $2^{ième}$ instance au top 2 car le 1^{ier} job de τ_2 s'est terminé entre les tops 1 et 2.

$\mathcal{S}$	*size*	T_i	O_i	C_i	D_i	$\mathcal{R}$	$\mathcal{C}$	Core
τ_1	121	2	0	1	2	$\tau_{2.0} \rightarrow \tau_{1.1}$	$\tau_{2.0}.o \rightarrow \tau_{1.1}.i$	0
τ_2	251	2	0	1	2	$\tau_{1.0} \rightarrow \tau_{2.0}$	$\tau_{1.0}.o \rightarrow \tau_{2.0}.i$	1
τ_3	304	4	0	2	4	$\tau_{1.0} \rightarrow \tau_{3.0}$	$\tau_{1.0}.p \rightarrow \tau_{3.0}.i$	2

Table 4.3 – Exemple d'ensemble de tâches



(a) Comportement idéal (b) Comportement avec tick-gap

Figure 4.5 – Ordonnancement dirigé par le temps

Cette approche marche très bien sur des multi-cœurs dont les horloges locales sont parfaitement synchronisées et dont la cohérence de cache est assurée. Du fait (1) de la précision imparfaite des horloges et (2) des temps de communication non négligeables entre cœurs, l'implantation de l'ordonnanceur doit être modifiée. Pour assurer que les cœurs ont une vue cohérente de l'état des autres tâches, l'ordonnanceur impose un décalage (*gap*) entre le top d'horloge et le début d'un job. Cette idée est inspirée du concept *sparse time* de [Kop92]. Le choix de la largeur du *gap*, noté t_{gap}, dépend de deux caractéristiques de la plate-forme :

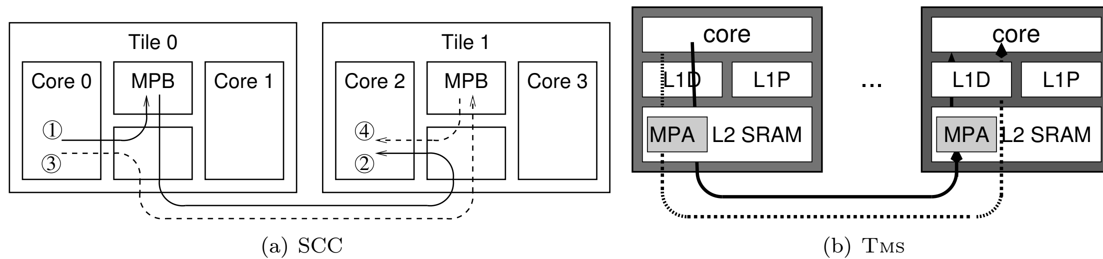
1. la précision d'horloge Δ_{clock} ;
2. le temps maximal de communication entre deux cœurs t_{comm}.

La figure 4.5(b) montre l'ordonnanceur modifié pour le même ensemble de tâches. Les tops d'horloge sur les cœurs sont légèrement désynchronisés mais avec une distance deux à deux $\leq \Delta_{clock}$. Après chaque top d'horloge il y a une attente de t_{gap} pour être assuré de la vision cohérente. La tâche τ_1 termine encore entre 0 et 1 et elle envoie une donnée à τ_2 en t_{comm} unités de temps. Cette donnée arrive après le top 1 sur le cœur 2, pendant l'attente du gap. Ainsi, lorsque τ_2 commence la donnée est là. Il faut donc que les tops d'horloge soient beaucoup plus grands que t_{gap} pour pouvoir ordonnancer des tâches. On prend la plus petite valeur possible pour t_{gap} à savoir $t_{gap} = \Delta_{clock} + t_{comm}$.

4.3.3 Gestion des communications

Les deux plates-formes considérées (SCC et Tms) peuvent être vues comme des architectures distribuées avec leur mémoire locale. Les communications entre tâches sont réalisées de manière explicite par passage de message. Il y a deux approches standards pour placer un message : (1) dans le MPA du

lecteur/récepteur (type *push*) ou (2) dans celui de l'écrivain/producteur (type *pull*). Les performances dépendent de l'application : s'il y a beaucoup de broadcast le pulling est meilleur ; en revanche si les délais sont courts, l'envoi *push* est plus performant car non bloquant et uni-directionnel tandis qu'une récupération *pull* peut nécessiter des requêtes supplémentaires.



(a) SCC (b) TMS

FIGURE 4.6 – Patrons de communication push/pull

La figure 4.6(a) illustre les communications pull/push sur le SCC. Deux tâches s'exécutent, l'une sur le cœur 0 et l'autre sur le 2, et il y a une communication de 0 → 2. Les lignes continues montrent une communication de type *pull*. A l'étape ①, le cœur 0 écrit dans son MPA local et à l'étape ②, le récepteur sur le cœur 2 fait une lecture distante sur la tuile 0. A l'inverse, une communication *push* est illustrée par le trait pointillé. A l'étape ③, la tâche sur le cœur 0 fait une écriture distante et à l'étape ④, le récepteur lit son MPA local.

La figure 4.6(b) illustre les communications pull/push sur le TMS. Le fonctionnement est assez similaire au SCC : en pointillé on retrouve le *pull* avec une écriture locale et en continu un *push* avec une écriture distante.

4.3.4 Détails sur les librairies

Programmation bare-metal SCC Nous avons réutilisé un environnement bare-metal appelé BAREMICHAEL [ZB12] qui a été étendu par Johannes Scheller, étudiant master d'Eric Noulard [Sch12]. Cette extension contenait des primitives pour le passage de message, des moyens de compilation MIMD et l'implantation de la barrière de synchronisation. Le modèle d'exécution implanté est le 3.

Programmation bare-metal TMS La librairie bare-metal a été développée pour l'implantation du modèle AER décrit dans la partie 3.4. Ce modèle repose notamment sur une configuration précise des mémoires où tous les L2 sont SRAM et le MPA est réservée aux communications entre tâches. Pour les communications I/Os, les données passent par la NAND de la puce et sont représentées par des A et E.

Enfin, un dispatcher a été « *hard codé* ». Considérons un ensemble de tâches géométriques, c'est-à-dire donc les périodes sont multiples les unes des autres, tel que T_0 est la plus petite période (pgcd des T_i) et tel que $\sum C_i \leq T_0$. On peut donc définir statiquement une séquence sur T_0, sachant que les tâches qui ne sont pas sur T_0 ne s'exécutent pas à chaque fois. La structure est donnée dans la figure 4.7 dans *static_sched* : on donne la date de démarrage de la tâche, l'adresse de la tâche et le modulo d'exécution. Ensuite, ce tableau est parcouru toutes les périodes T_0.

4.4 Conclusion

Les outils de configuration automatiques sont indispensables face à la complexité des applications, des plates-formes matérielles et des règles d'exécution. Les méthodes formelles, comme les approches par contraintes (ex. logiciel ILOG OPL) ou SMT (Satisfiability Modulo Theories, ex. logiciel Z3), sont aujourd'hui capables de traiter des problèmes de grandes tailles et notamment des systèmes industriels comme nous l'avons montré au travers de nos exemples. L'utilisation de méthodes formelles dans le développement de systèmes critiques est un avantage certain, à la fois pour la certification, puisqu'elles sont reconnues comme un argument réel dans le supplément méthodes formelles DO 333 [RTC11b] du DO 178C [RTC11a], mais également pour les coûts de vérification et validation.

Les exécutifs ou hyperviseurs critiques temps réel existants pour les multi/pluri-cœurs COTS ne répondent pas encore aux besoins des applications avioniques de type contrôle/commande. C'est la raison

```
typedef struct  static_sched {
  int t;
  void * task;
  int modulo;
} static_sched;

const static_sched mysched [N]={(t0,&task0,mod0), (t1,&task1,mod1), ...};

for(i=0;i<ITER;i++) {
  for(j=0;j<N;j++) {
    t1 += mysched [j]-> t;
    active_wait_until(t1);
    if(i% mysched [j]-> modulo ==0){
      mysched [j]-> task();
    }
  }
  t1 += T0;
  active_wait_until(t1);
}
```

Figure 4.7 – Exemple de codage d'un séquenceur

pour laquelle nous avons codé des librairies bare-metal. Notre objectif n'est pas de nous investir dans le développement d'exécutifs complexes mais de programmer des besoins ponctuels en attendant de trouver des supports répondant à nos attentes. Dans le projet Dreams, une contribution sera de programmer des services bare-metal de surveillance et d'adaptation pour supporter des défaillances temporelles sur le Freescal T4240. Ces librairies seront intégrées dans l'hyperviseur Xtratum [CRM10b] et deviendront ainsi des services à part entière.

Chapitre 5

Exécution prédictible de systèmes à criticité mixte

L'ensemble des travaux présentés jusqu'à présent montrait comment porter de manière prédictible une application unique sur une plate-forme. Or une application seule ne peut justifier l'utilisation d'un multi/pluri-cœurs à 48 ou 256 cœurs. Il faut donc mutualiser les cibles entre plusieurs applications. Regrouper les plus critiques ne suffit pas non plus à remplir les capacités de calcul de la plate-forme (et est déconseillé en terme de sûreté de fonctionnement car le processeur deviendrait un point de défaillance unique), de ce fait il faut intégrer des applications qui ne sont pas toutes soumises aux mêmes contraintes de sécurité. Ce chapitre présente des travaux plus récents sur l'implantation prédictible d'applications dans un environnement à « *criticité mixte* ».

5.1 Contexte

Intégrer des applications de nature diverse sur une même plate-forme embarquée est possible grâce aux principes fondamentaux de l'*avionique modulaire intégrée* (Integrated Modular Avionic - IMA), adoptés par Airbus depuis l'A380 et par Boeing depuis le B777. Ces architectures sont apparues dans les années 90 pour exécuter un ensemble d'applications avioniques partageant des ressources de calcul, appelées modules, et communiquant par un réseau partagé connecté à un ensemble de capteurs/actionneurs par l'intermédiaire de passerelles. Le modèle d'exécution des modules IMA est défini par le standard ARINC 653 [Aer97] tandis que le modèle de communication est celui de l'ARINC 664 (partie 7) [Aer05]. L'objectif de ces standards est d'assurer un niveau élevé de sûreté de fonctionnement, en particulier en garantissant à la fois la ségrégation des fonctions avioniques et un certain degré de prédictibilité temporelle. Les architectures IMA actuelles embarquent des mono-processeurs car garantir les mêmes propriétés sur un multi/pluri-cœurs, du fait de la grande variabilité temporelle et de la forte interdépendance des comportements, reste encore un problème ouvert et un axe très actif de recherche.

Dans la communauté temps réel, on ne parle pas d'architectures IMA mais de *systèmes à criticité mixte* (mixed-critical systems) [Ves07]. Le niveau de criticité dépend des conséquences d'une défaillance de l'application sur le système global. Nous reprenons la nomenclature avionique pour définir ces niveaux : on parle de *niveau d'assurance de développement* (Design Assurance Level - DAL) [SAE10] allant du plus contraint A au moins contraint E. Puisque nous nous intéressons à des questions temporelles, notre modèle de faute est le *non respect des échéances temporelles.* Nous dirons dans la suite que les applications *temps réel dur* (ou de haute criticité) sont de type A, B où C tandis que celles *temps réel mou* (ou moins critiques) sont de niveau D ou E.

5.1.1 Besoins et objectifs

Considérons $n+1$ tâches synchrones indépendantes $\mathcal{T} = \{\tau_C, \tau_1, \ldots, \tau_n\}$ telles τ_C est une tâche périodique de haute criticité (DAL A, B ou C), de période T_C et d'échéance D_C ; τ_i sont des tâches moins critiques (DAL D or E). On souhaite exécuter ces tâches sur un multi-cœurs en appliquant un ordonnancement partitionné.

Quand toutes les tâches s'exécutent en parallèle, le WCET de τ_C est estimé au dessus de son échéance

du fait des accès aux ressources partagées et de l'hypothèse de congestion totale supposée par les analyses de WCET. Ce scénario est appelé *charge maximale* (max) et est illustré dans la figure 5.1.a., où $\mathsf{WCET}_{\mathsf{max}} > D_C$.

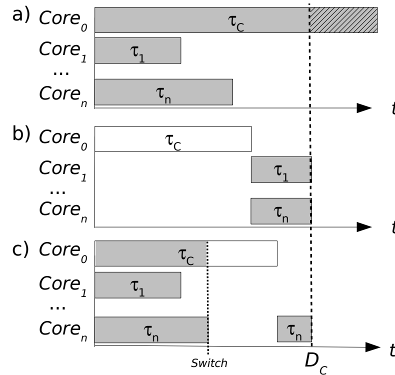


FIGURE 5.1 – Scénarios de scénarios d'ordonnancement

Une solution sûre consiste alors à exécuter la tâche critique seule, ce qui élimine toute possibilité d'accès concurrent, puis exécuter les tâches non critiques s'il reste du temps disponible. On parle alors de scénario *isolation* (iso). Le WCET calculé est significativement plus petit et la tâche critique respecte son échéance i.e. $\mathsf{WCET}_{\mathsf{iso}} \leq D_C$, ce qui est illustré dans la figure 5.1.b. Dans ce cas, les contraintes sont respectées mais les ressources sont sous utilisées.

Notre approche est un entre-deux de ces solutions extrêmes. Les tâches moins critiques peuvent s'exécuter en parallèle tant que la tâche critique est certaine de terminer avant son échéance temporelle, comme illustré figure 5.1.c. Pour ce faire, un superviseur en ligne (ou *run-time controller*) surveille régulièrement l'exécution en cours de τ_{C} et vérifie si les interférences générées par les τ_i sont acceptables. S'il y a un risque de dépassement de l'échéance, le contrôleur suspend l'exécution des tâches non critiques et les relance quand τ_{C} se termine. De cette façon, τ_{C} respecte toujours son échéance et l'utilisation des ressources partagées est augmentée en acceptant l'exécution concurrente de toutes les tâches. L'idée principale de la méthodologie a été introduite dans [KBP13], la preuve de correction de l'algorithme est montrée dans [KBP14] et l'extension à plusieurs tâches critiques est détaillée dans [KPR14]. Ces travaux résultent de la collaboration des trois laboratoires toulousains (IRIT, LAAS, ONERA) et Thales dans le cadre du projet RTRA TOAST finançant le post-doctorat d'Angeliki Kritikakou (2013-2014) et du projet européen FP7 DREAMS.

5.1.2 Travaux connexes

Cette partie décrit des approches existantes pour implanter des systèmes à criticité mixte. Une vue d'ensemble détaillée est proposé par [BD14].

Gestion par des techniques d'ordonnancement Les premiers travaux sur l'ordonnancement de systèmes à criticité mixte visaient les architectures mono-processeurs (ex. [Ves07 ; BV08 ; BLS10 ; BB13]), résultats qui ne sont donc pas directement applicables aux multi/pluri-cœurs du fait de la non *composabilité temporelle* des comportements [CFG10]. La composabilité temporelle signifie que l'on peut calculer les WCET des tâches indépendamment et en déduire le WCET en concurrence qu'elles que soient les configurations.

Plusieurs approches ont ensuite été définies pour les multi-cœurs, supposant toutes l'ensemble de tâches ordonnançable au niveau haute criticité, ce qui signifie que pour toute tâche τ, on a $\mathsf{WCET}_{\mathsf{criticit\acute{e}\ haute}}(\tau) \leq D_\tau$. Par exemple, dans [BA07], toutes les tâches sont ordonnancées avec la politique *Earliest Deadline First for Hard real-time, Soft real-time and Best effort* (EDF-HSB) et si un cœur finit plus tôt, le temps est ré-alloué aux tâches moins critiques. Un autre exemple est l'ordonnancement à deux niveaux proposé par [ABB09] et étendu dans [MEA10]. Les tâches critiques sont d'abord ordonnancées en *isolation*, puis les moins critiques s'exécutent. L'approche est implantée dans LITMUS^{RT} [HKM12].

L'idée partagée par les autres approches, telles que [LB12 ; BCLS14 ; Pat12], est d'associer plusieurs valeurs de WCET à chaque tâche et de considérer plusieurs ordonnancements, un par niveau de criticité. Ces différents WCET dérivent d'ensemble de cas de test comme expliqué dans [BB11] : pour les niveaux

de criticité élevés, les mesures sont plus intensives et explorent davantage de comportements, produisant des valeurs plus élevées (et donc plus sûres). Initialement, les tâches ont leur WCET le plus faible. Les algorithmes décrits dans [LB12 ; BCLS14] reposent sur une généralisation d'EDF avec des échéances virtuelles (EDF-VD). A l'exécution, si les tâches critiques n'ont pas signalé leur terminaison avant une date pré-définie, alors on passe dans un niveau de criticité plus élevé. Les tâches moins critiques sont éliminées. Dans [FB13], les auteurs ne jettent plus les tâches moins critiques mais les suspendent et les relancent lorsque le système peut retourner à un ordonnancement de niveau de criticité plus faible. L'approche présentée dans [BF11] considère des systèmes à criticité mixte ordonnancés de manière dirigée par le temps où les WCET peuvent être mal estimés et dépasser leur valeur. Un contrôleur en-ligne détecte ces dépassements et passe en mode de séquencements pré-calculés.

Notre approche est complémentaire à celles mentionnées précédemment : nous considérons deux valeurs de WCET dépendant des modes d'exécutions sur la plate-forme, isolation ou concurrent, et non en fonction de niveau de criticité. Nos estimations sont toujours sûres car reposant sur une analyse statique. Enfin, la suspension des tâches moins critiques n'arrive pas à une date pré-définie mais dès qu'un problème semble survenir.

Gestion par des techniques de supervision en ligne Plusieurs approches proposent de ré-allouer les ressources aux tâches moins critiques en surveillant leur utilisation, en particulier leurs accès mémoire. Dans [NPB14], une analyse préliminaire d'utilisation des ressources par tâche est faite, ce qui permet ensuite de partager l'accès aux ressources par un ordonnancement hors-ligne. Un contrôleur en-ligne vérifie que les tâches respectent leur contrat d'utilisation, sinon il les suspend. Les auteurs étendent leur approche dans [NP13] en acceptant que le partitionnement des ressources soit modifié en-ligne, lorsque les ressources sont sous-utilisées. Une autre approche, proposée dans [YYP13], permet de réserver des ressources pour les tâches grâce à un contrôleur en-ligne qui régule les accès à la mémoire partagée et assure l'isolation spatiale. Un profiling hors-ligne est décrit dans [MDB13] afin de déterminer quelles sont les pages mémoire les plus accédées. Cette information est ensuite utilisée pour modifier la position des variables en cache de manière à minimiser les interférences avec la mémoire. Enfin, un contrôleur en-ligne matériel de gestion de la mémoire, décrit dans [DS14], ne s'exécute que pendant des moments où les cœurs sont libres, ce qui assure que le superviseur n'a pas d'impact sur l'exécution.

Notre approche n'est pas basée sur la surveillance des accès aux ressources partagées mais sur le comportement en cours de la tâche critique, ce qui nous donne implicitement une information sur les interférences.

5.2 Approche

Angeliki Kritikakou a travaillé pendant son post-doctorat sur la formalisation du contrôleur en-ligne et son implantation sur le TMS.

5.2.1 Idée générale

La méthodologie, montrée figure 5.2, combine (1) une phase d'analyse hors-ligne pendant laquelle un certain nombre d'informations sont pré-calculées puis utilisées à l'exécution par (2) le contrôleur en ligne qui gère les tâches sur la plate-forme.

Phase hors-ligne On considère $n+1$ tâches indépendantes synchrones $\mathcal{T} = \{\tau_\mathrm{C}, \tau_1, \ldots, \tau_n\}$ où τ_C est une tâche de haut niveau de criticité et τ_i sont moins critiques. Ces tâches sont exécutées sur un multi-cœurs selon un ordonnancement partitionné et une unique tâche s'exécute sur chaque cœur.

La tâche critique est représentée par un ensemble de *graphes de flot de contrôle étendus* (Extended Control Flow Graph - ECFG), où un ECFG est un graphe de flot de contrôle avec des *points d'observation*. Nous introduisons ces points pour permettre au contrôleur de surveiller en-ligne le comportement temporel de la tâche critique et d'interrompre/reprendre les moins critiques. A chaque point d'observation, la *condition de sûreté*, donnée dans l'équation 5.1, vérifie s'il est toujours sûr de continuer l'exécution de τ_C avec le scénario *charge maximale*, et passe en scénario *isolation* en interrompant les autres cœurs sinon. Il faut donc pré-calculer les informations de structure et temporelles nécessaires au contrôleur. L'approche a été implantée sur le TMS avec un contrôleur logiciel, c'est-à-dire que chaque point d'observation de τ_C est raffiné par un morceau logiciel qui teste la condition de sûreté et interrompt les cœurs.

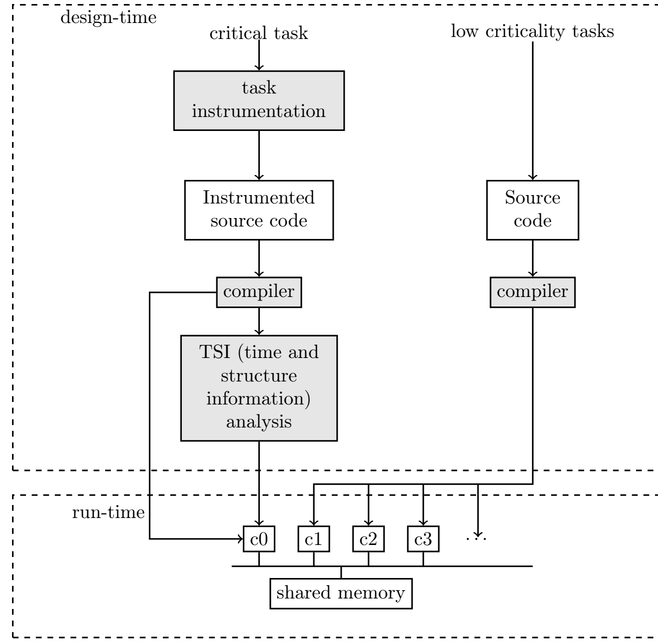


FIGURE 5.2 – Description de la méthode de supervision

Phase en-ligne Le système commence l'exécution en *charge maximale*. Le contrôleur passe du scénario *charge maximale* à *isolation* au point d'observation x lorsque la *condition de sûreté* n'est pas satisfaite :

$$\mathsf{RWCET_{iso}}(x) + \mathsf{W_{max}} + t_{\mathrm{RT}} \leq D_{\mathrm{C}} - \mathsf{ET}(x) \tag{5.1}$$

où $\mathsf{RWCET_{iso}}(x)$ est le WCET restant de la tâche critique dans le scénario en *isolation* en x, $\mathsf{W_{max}}$ est le WCET en *charge maximale* jusqu'au prochain prochain point d'observation, t_{RT} est le temps total d'exécution du mécanisme du contrôleur en-ligne et $\mathsf{ET}(x)$ est le temps d'exécution réel de τ_C jusqu'au point x. Plus précisément t_{RT} est la somme de : 1) t_{Mon} (overhead pour surveiller le temps d'exécution), 2) t_{Cnt} (WCET du contrôle), and 3) t_{SW} (WCET pour changer de scénario).

Théorème 2. *Si* $\textit{WCET}_{iso/cri} \leq D_C$*, pour toute exécution avec le contrôleur, alors* τ_C *respecte toujours son échéance.*

Extension à plusieurs tâches critiques Nous avons ensuite étendu l'approche pour gérer plusieurs tâches critiques lesquelles peuvent également avoir des dates de réveil non nulles. A nouveau l'ordonnancement est partitionné et une seule tâche est allouée sur chaque cœur. Le modèle de système consiste donc en $p + n$ tâches indépendantes $\mathcal{T} = \{\tau_{\mathrm{C}_1}, \ldots, \tau_{\mathrm{C}_\mathrm{p}}, \tau_1, \ldots, \tau_n\}$ où les $\tau_{\mathrm{C_j}}$ sont des tâches périodiques de haute criticité avec pour période T_{C_j}, échéance D_{C_j} et date de réveil O_{C_j} ; les τ_i sont les tâches moins critiques.

On ajoute un nouveau scénario *critiques uniquement* aux deux existants *charge maximale* et *isolation*. Pour ce nouveau scénario, le WCET d'une tâche critique est noté $\mathsf{WCET_{cri}}$ tel que

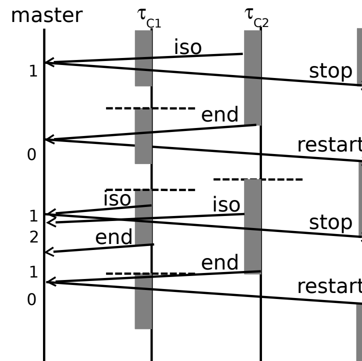
$$\mathsf{WCET_{iso}} \leq \mathsf{WCET_{cri}} \leq D_C < \mathsf{WCET_{max}}$$

Cela suppose que l'outil d'analyse statique soit capable de calculer ce $\mathsf{WCET_{cri}}$. A l'exécution, chaque tâche est surveillée par son propre mécanisme de contrôle, lequel calcule le WCET restant en *critiques uniquement*, vérifie la condition de sûreté pour décider si les tâches moins critiques doivent être suspendues afin de garantir le respect des échéances. La condition de sûreté est similaire :

$$\mathsf{RWCET_{cri}}(x) + \mathsf{W_{max}} + t_{\mathrm{RT}} \leq D_{\mathrm{C}} - \mathsf{ET}(x) \tag{5.2}$$

où $\mathsf{RWCET_{cri}}(x)$ est le WCET restant de la tâche critique dans le scénario en *critiques uniquement* en x. Les tâches hautement critiques ne sont plus responsables de la suspension τ_i. Elles envoient leurs requêtes à une nouvelle entité, appelée le *maître* qui a une vue globale de la situation. Le maître est en charge de collecter les requêtes des tâches hautement critiques, de suspendre et redémarrer les tâches moins

critiques. La suspension a lieu si au moins une des tâches critiques le demande tandis que le redémarrage ne peut se faire qu'à condition que toutes les tâches critiques l'acceptent.



La figure ci-dessus illustre le comportement du mécanisme de supervision sur un exemple composé de deux tâches critiques. Dans un premier temps, la condition de sûreté de τ_{C_2} est violée et la tâche demande le passage en *critiques uniquement* au maître. Ce dernier met son compteur interne des requêtes actives à 1 et suspend les tâches moins critiques. τ_2 informe ensuite le maître de la fin de son exécution. Comme τ_{C_1} n'a pas demandé de passage en scénario *critiques uniquement*, il n'y a pas de danger pour l'échéance de τ_{C_1} et le maître redémarre les tâches moins critiques (le compteur des requêtes actives est alors à 0). Plus tard, τ_{C_1} suivi de τ_{C_2} demande un arrêt des interférences des tâches moins critiques (le compteur des requêtes actives est alors à 2). Le maître redémarre les tâches uniquement lorsque τ_{C_1} et τ_{C_2} ont terminé leurs instances.

5.2.2 Représentation d'une tâche critique

Graphe de flot de contrôle étendu

Une tâche critique doit respecter la syntaxe générale décrite table ci-dessous, laquelle couvre un large spectre d'applications.

Syntax rules		
term	: :=	<constant> | <variable>
expr	: :=	<term> | <term> <operator> <term> | <unary-expr>
unary-expr	: :=	<variable> <unary-operator> | <unary-operator> <variable>
cond-expr	: :=	<expr> <conditional-operator> <expr>
assignment	: :=	<unary-expr> <assignment-operator> <expr>
instruction	: :=	<assignment> | <unary-expr> | <> ;
stat	: :=	<instruction> | <stat> ;<stat> | if (<cond-expr>) then <stat1> else <stat2> | for (expr1 ; cond-expr ; expr2) <stat> | <function-call> ;
function-call	: :=	<return-type> functionName(<parameter-list>) <stat> return <expr-return> ;
program	: :=	<function-call>

Définition 17. *Une* tâche critique τ_C *est un ensemble de fonctions* $\mathcal{S} = \{F_0, F_1, ..., F_n\}$ *où* F_0 *est la fonction principale (*main*).*

Le binaire obtenu après compilation (pour des raisons de traçabilité, on choisit de n'appliquer aucune optimisation) peut être représenté par un ensemble de *graphes de flot de contrôle* (CFGs) [CHW02]. On prend en entrée ces CFGs afin d'y insérer des points d'observation.

Définition 18 (Graphe de flot de contrôle étendu)**.** *Un ECFG (graphe de flot de contrôle étendu - extended control flow graph) est un graphe de flot de contrôle contenant des* points d'observation*. Un point d'observation est une position où le contrôleur en-ligne est exécuté. L'ECFG associé à une fonction* F *est un graphe dirigé* $G = (V, E)$ *constitué de :*

1. *un ensemble fini de nœuds* V *composé de 5 ensembles disjoints* $V = \mathcal{N} \cup \mathcal{C} \cup \mathcal{F} \cup \{IN\} \cup \{OUT\}$ *où :*
   — $N \in \mathcal{N}$ *représente une instruction binaire ou un bloc d'instructions binaires (fig. 5.3(a)),*
   — $C \in \mathcal{C}$ *représente un bloc d'instructions binaires d'une condition de branchement (fig. 5.3(b)),*
   — $F_i \in \mathcal{F}$ *représente un bloc d'instructions binaires d'un appel de fonction* F_i *et relie le nœud avec l'ECFG de la fonction* F_i *(fig. 5.3(c)),*

- *IN est le nœud d'entrée (fig. 5.3(d)),*
- *OUT est le nœud de sortie (fig. 5.3(e)).*
- *tous les nœuds $v \in V$ ont un unique point d'observation avant l'exécution de la première instruction du bloc (un point d'observation est écrit en minuscule) ;*
- start *est un point d'observation spécial qui a lieu avant le début de l'exécution.*

2. *un ensemble fini d'arêtes $E \subseteq V \times V$ décrivant le flot de contrôle entre les nœuds.*

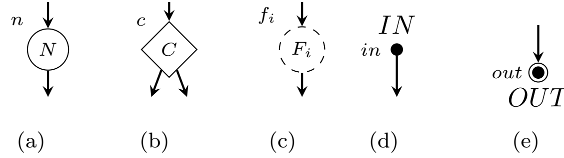


FIGURE 5.3 – Représentation schématique des nœuds terminaux de la grammaire

Définition 19 (Syntaxe d'un ECFG)**.** *La syntaxe permettant de décrire un ECFG est définie par la grammaire de graphe suivante. On considère un unique nœud non-terminal B et tandis que ceux de la définition des ECFGs sont des terminaux. Les règles de la grammaire sont :*

- *Une fonction F_i a exactement un unique nœud d'entrée et un unique nœud de sortie. Les règles s'appliquent sur le non-terminal B (fig. 5.4(a)).*
- *Un non-terminal B est dérivé comme suit :*
  1. *le nœud vide (fig. 5.4(c)),*
  2. *un nœud simple N (fig. 5.4(d)),*
  3. *une séquence, i.e. concaténation de non-terminaux (fig. 5.4(e)),*
  4. *un composant de type* if-then-else, *i.e. concaténation d'une condition de branche C et de deux chemins en exclusion mutuelle terminant par le même non-terminal (fig. 5.4(f)),*
  5. *une boucle, i.e. concaténation d'une condition de boucle C avec deux chemins en exclusion mutuelle, l'un avec le nœud vide et l'autre étant la répétition du corps de boucle (fig. 5.4(g)),*
  6. *un appel de fonction F_i (fig. 5.4(h)).*
- *Durant chaque dérivation, les chemins entrants dans B sont* groupés *et le sortant sera assemblé avec tous les entrants du bloc suivant.*

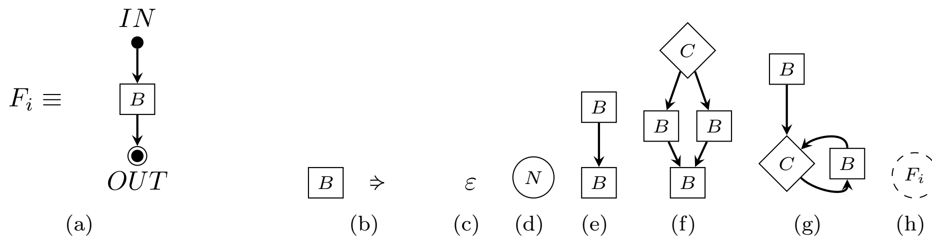


FIGURE 5.4 – Représentation schématique des règles de récriture de la grammaire

Chaque fonction est représentée par un ECFG et la tâche critique peut être vue comme un ensemble d'ECFGs disjoints. L'appel de fonction lie les ECFGs disjoints associés aux différentes fonctions.

Définition 20 (Graphes disjoints)**.** *Deux ECFG $G_i = (V_i, E_i)$ et $G_j = (V_j, E_j)$ sont disjoints si $V_i \cap V_j = \emptyset$.*

Exemple 28. *La figure 5.5 décrit une tâche composée de deux fonctions $\mathcal{S} = \{F_0, F_1\}$. Dans la figure 5.5(a), F_1 est séquentielle tandis que dans la figure 5.5(b) F_1 est récursive. Les fonctions récursives sont exprimées grâce à une condition de branchement qui distingue le chemin récursif du cas de base.*

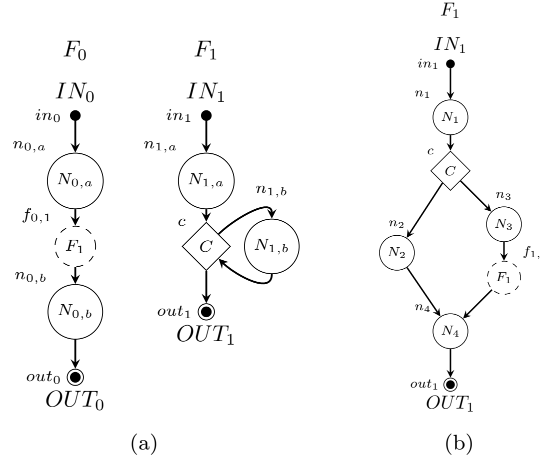


FIGURE 5.5 – ECFGs disjoints associés à $\mathcal{S} = \{F_0, F_1\}$

Exécution

Une exécution de la tâche critique peut être vue comme une suite de points d'observation. On définit une *marche* comme une exécution dans un unique ECFG et une exécution comme un chemin traversant plusieurs ECFGs.

Définition 21 (Marche)**.** *Une marche $W = (in, v_1, \ldots, v_n, out)$ dans un ECFG est définie par une suite de points tels que $(V_i, V_{i+1}) \in E$, $(IN, v_1) \in E$ et $(v_n, OUT) \in E$ (on rappelle que les points sont notés en minuscule et les nœuds associés en majuscule).*

Définition 22 (Exécution)**.** *Considérons la tâche $\mathcal{S} = \{F_0, ..., F_n\}$ et les ECFGs associés $G_0, \ldots G_n$. Une exécution P est une liste de points associés aux nœuds $\in V_0 \cup \ldots \cup V_n$ obtenus en insérant des marches à travers les visites d'ECFG. Une exécution est donc obtenue inductivement comme suit :*

- *Initialement, P=W_o, où W_o est une marche sur l'ECFG de F_0, $W_o = (in_0, v_1, \ldots, out_0)$.*
- *Soit $P = (in_0, \ldots, v_i, v_{i+1}, \ldots, out_0)$. Si $V_i = F_k$ et $V_{i+1} \neq IN_k$, alors P est récrit en $P = (IN_0, \ldots, v_i, W_{F_k}, v_{i+1}, \ldots, OUT_0)$ où W_{F_k} est une marche sur G_k.*

Exemple 29. *Reprenons la figure 5.5(a), $P = (in_0, n_{0,a}, f_1, in_1, n_{1,a}, c, n_{1,b}, c, out_1, n_{0,b}, out_0)$ est une exécution de la tâche. De la même manière, une exécution pour la fonction récursive figure 5.5(b) est $P = (in_0, n_{0,a}, f_1, in_1, n_1, c, n_3, f_1, in_1, n_1, c, n_2, n_4, out_1, n_4, out_1, n_{0,b}, out_0)$.*

5.3 Phase hors-ligne

Le contrôleur en-ligne a en charge de surveiller les points d'observation et les temps d'exécution réels. Il faut donc être en mesure de surveiller les points et pour ce faire nous avons choisi d'instrumenter les tâches. De plus, calculer dynamiquement la condition de sûreté serait trop coûteux, nous avons donc opté pour le pré-calcul un certain nombre d'information.

5.3.1 Instrumentation d'une tâche critique

Le contrôleur doit surveiller le comportement de la tâche critique. Il y a deux principales approches :

1. matérielle : en surveillant les instructions avec un composant hardware;
2. logicielle : en ajoutant une instrumentation logicielle. Notre cible étant un multi-cœurs COTS ne permettant pas d'observer matériellement les comportements, nous avons choisi cette deuxième option.

Définition 23 (Contrôleur logiciel)**.** *Un* contrôleur logiciel *(fig. 5.6) consiste en :*

1. *une condition C_R dont la sémantique est :*
   - *C_R est true $\Longleftrightarrow$ exécution en scénario maximale charge,*

— C_R *est false* $\iff \overline{C_R} \iff$ *exécution en scénario* isolation *ou* critiques uniquement*,*
— $C_R = $ *true au point d'observation* start*.*

2. un comportement adéquat :
— *lorsque* C_R *est vrai, un appel de fonction* F_D *qui : 1) calcule* $\mathsf{RWCET}_{\mathsf{iso/cri}}(x)$ *(nœud* N_C*), vérifie la condition de sûreté (nœud* C_D*) et éventuellement passe en mode* isolation *ou* critiques uniquement *(nœud* N_S*), mettant ainsi* C_R *à false et 2) est exécuté au point d'observation* $v_i \in V \setminus \{IN, OUT\}$*,*
— *lorsque* $\overline{C_R}$ *est vrai, un nœud* N_E *qui : 1) envoie un événement pour redémarrer les tâches moins critiques et 2) est exécuté au point d'observation* $v_i = OUT_0$ *(i.e. la fin de* τ_C*).*

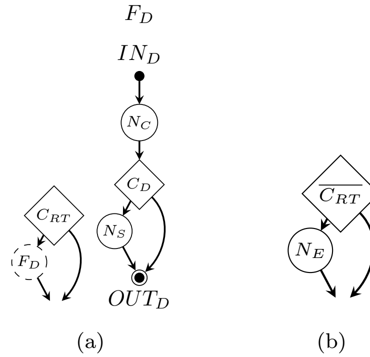


FIGURE 5.6 – Grammaire du contrôleur logiciel

Exemple 30. *Reprenons les ECFGs théoriques de la figure 5.5(a), ils sont instrumentés dans la figure 5.7(a). Dans la suite, un* ECFG instrumenté *représente un ECFG dont les points sont implantés par un contrôleur logiciel.*

5.3.2 Analyseur hors-ligne

L'analyseur hors-ligne calcule les informations de structure et temporelles nécessaire pour le contrôleur en-ligne. Sa description est donnée dans la figure 5.7(b).

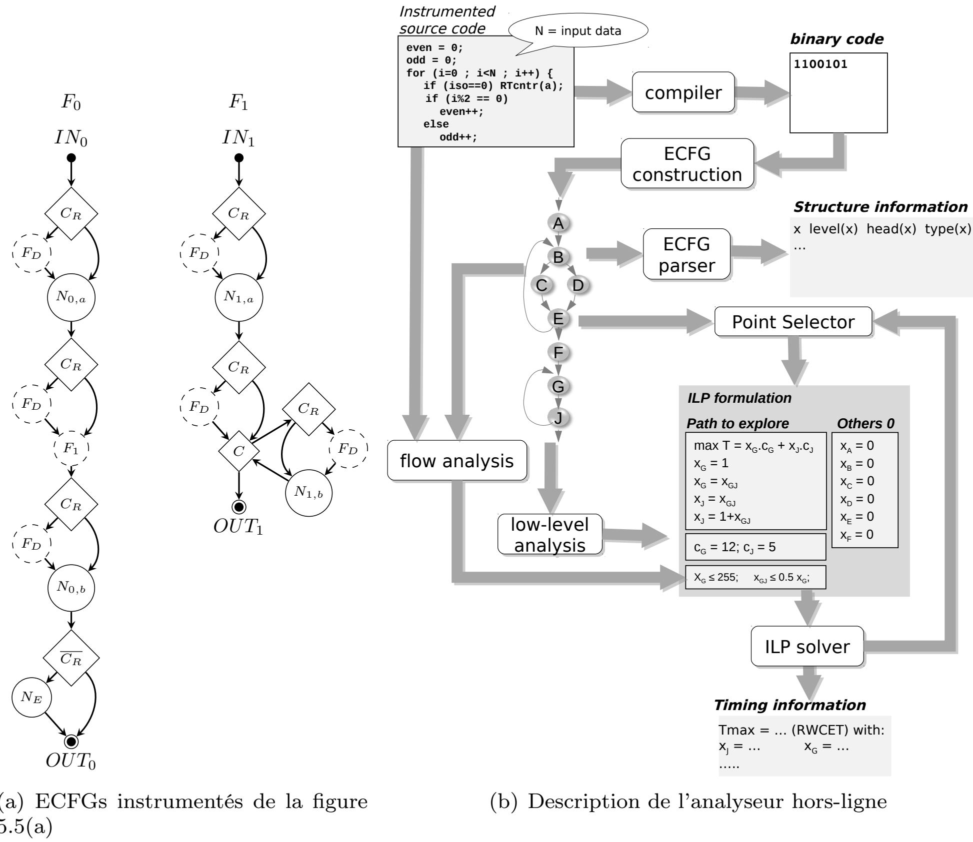
Informations structurelles Un point d'observation peut être visité plusieurs fois, à cause des boucles et des appels de fonction. Il faut donc pouvoir distinguer :

1. le *niveau d'imbrication* qui définit la profondeur d'un point dans un ECFG ;
2. le *point supérieur* qui donne le point appelant d'une fonction ou la tête de boucle ;
3. le *type* qui détermine si un point réalise un appel de fonction ou est le point de retour d'une fonction appelée.

Nous formalisons ces trois concepts dans la suite.

Définition 24 (Niveau d'imbrication)**.** *Le niveau d'imbrication, noté* level(x)*, du point d'observation* x *est défini comme suit :*
— start *: le niveau du point d'initialisation est toujours* level(start)=0 *;*
— in *et* out *: le niveau des points d'entrée et sortie est toujours* level(in)=1 *et* level(out)=1.
— *séquence : le niveau de deux points d'observation* b_1 *et* b_2 *en séquence est identique* level(b_1)=level(b_2) *;*
— if-then-else *: le niveau des 4 points* c*,* b_t*,* b_f *et* b_o *est identique* level(c)=level(b_t) =level(b_f)=level(b_o) *;*
— boucle *: le niveau du bloc initial* b_i *est égal au niveau de la condition* c*, i.e.* level(b_i)=level(c)*. Le niveau du corps de boucle est incrémenté de 1 par rapport à la condition* c *:* level(b) = level$(c) + 1$.

Définition 25 (Point supérieur)**.** *On considère la tâche* $\mathcal{S} = \{F_0, ..., F_n\}$*, les ECFGs associés* $G_0, \ldots G_n$ *et une exécution* $P = (in_0, v_1, \ldots, out_0)$*. Le point supérieur,* head(x)*, est défini comme suit :*
— head(in_0)=start



(a) ECFGs instrumentés de la figure 5.5(a)

(b) Description de l'analyseur hors-ligne

FIGURE 5.7 – Instrumentation hors-ligne de la tâche critique

- *on suppose qu'on a calculé* $\text{head}(v_i)$.
  - *si* $v_{i+1} = in_k$, *alors* $\text{head}(v_{i+1}) = v_i$
  - *si* $v_i = out_k$, *alors* $\text{head}(v_{i+1}) = \text{head}(\text{head}(v_i))$
  - *si* $(v_i, v_{i+1}) \in E_k$ *(i.e. dans la même ECFG) :*
    - *si* v_i *est une condition de boucle,* $\text{head}(v_{i+1}) = v_i$
    - *sinon* $\text{head}(v_{i+1}) = \text{head}(v_i)$

Définition 26 (Type)**.** *Le type d'un point d'observation* x*, noté* $\text{type}(x)$*, appartient à l'ensemble* $\{F_EN, F_EX, F_NX, -\}$ *avec :*
- *F_EN : appel de fonction ;*
- *F_EX : retour de fonction dans l'ECFG appelant ;*
- *F_NX : retour de fonction suivi d'un appel immédiat de fonction ;*
- *– : autre.*

Considérons l'exécution P=$(in_0, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, out_0)$*, le* type *(x) est défini comme suit :*
- *si* $v_i = f_k$
  - *si* $v_{i-1} = out_m$, $\text{type}(v_i) = F_NX$,
  - *sinon,* $\text{type}(v_i) = F_EN$.
- *si* $v_i \neq f_k$ *et* $v_{i-1} = out_m$, $\text{type}(v_i) = F_EX$,
- *sinon* $\text{type}(v_i) = -$

Exemple 31. *La table 5.1 fournit les informations structurelles pour les points d'observation de la figure 5.5.*

On stocke en mémoire les valeurs de *level*(x), *head*(x) et *type*(x) pour chaque point d'observation x. Les informations structurelles sont ensuite utilisées pour calculer les informations temporelles.

Informations temporelles Le calcul des WCET restant doit se faire si possible avec une méthode d'analyse statique de WCET. Il suffit d'étendre la méthode existante [WEE08 ; Roc11] : lors de la recherche des plus longs chemins, les blocs avant le point d'observation sont mis avec un temps d'exécution

Observation point x	*level* (x)	*type* (x)	*head* (x)	
Initialization				
start	0	-	-	
F_0				
in_0	1	-	*start*	
$n_{0,a}$	1	-	*start*	
$f_{0,1}$	1	*F_EN*	*start*	
$n_{0,b}$	1	*F_EX*	*start*	
out_0	1	-	*start*	
F_1 fig. 5.5(a)				
in_1	1	-	$f_{0,1}$	
$n_{1,a}$	1	-	$f_{0,1}$	
c	1	-	$f_{0,1}$	
$n_{1,b}$	2	-	c	
out_1	1	-	$f_{0,1}$	
F_1 fig. 5.5(b)				
			1st visit	2nd visit
in_1	1	-	$f_{0,1}$	$f_{1,1}$
n_1	1	-	$f_{0,1}$	$f_{1,1}$
c	1	-	$f_{0,1}$	$f_{1,1}$
n_3	2	-	c	
$f_{1,1}$	2	*F_EN*	c	
n_2	1	-	$f_{1,1}$	-
n_4	1	*F_EX*	$f_{1,1}$	$f_{0,1}$
out_1	1	-	$f_{1,1}$	$f_{0,1}$

Table 5.1 – Information structurelle pour la figure 5.5

nul. La faisabilité de l'approche a été testée dans l'outil Otawa. Nous calculons donc hors-ligne $\mathsf{RWCET_y}(x)$ pour tout point x et tout $y \in \{\mathsf{iso,cri,max}\}$. Bon nombre d'informations temporelles peuvent ensuite être déduites de $\mathsf{RWCET_y}(x)$ en tenant compte des points supérieurs *head*(x) et des corps de boucles.

Définition 27. *$d_{\text{head}(x)-x}$ est le temps maximal entre* head(x) *et* x.

$$d_{\text{head}(x)-x} = \mathit{RWCET}_{\mathit{iso/cri}}(\text{head}(x)) - \mathit{RWCET}_{\mathit{iso/cri}}(x)$$

Définition 28. *$w_{\text{head}(x)}$ est le temps maximal écoulé au niveau de la condition c entre deux itérations consécutives d'une boucle. Puisqu'on considère le pire scénario, il n'y a aucune variation temporelle entre deux itérations.*

$$w_c = \mathit{RWCET}_{\mathit{iso/cri}}(c,j) - \mathit{RWCET}_{\mathit{iso/cri}}(c,j+1), \forall j \leq n$$

Exemple 32. *Lorsqu'une fonction est appelée depuis plusieurs points, les différents chemins amenant à l'appel peuvent générer plusieurs WCETs restants. Cela peut donc introduire de la variabilité dans le calcul de $d_{head(x)-x}$ de la fonction appelée au point x. Par exemple, n_1 dans la figure 5.5(a) peut avoir différentes valeurs $d_{f_{0,1}-n_1}$ et $d_{f_{1,1}-n_1}$. Deux solutions existent : stocker toutes les valeurs possible ou stocker une information unique permettant le calcul d'une valeur sûre, i.e. le temps minimum observé pour l'appel de la fonction. Ainsi, dans la figure 5.5(b),* Otawa *calcule $\mathit{RWCET}_{\mathit{iso}}(f_{0,1}) = 50$, $\mathit{RWCET}_{\mathit{iso}}(n_1) = 48$, $\mathit{RWCET}_{\mathit{iso}}(f_{1,1}) = 45$, $\mathit{RWCET}_{\mathit{iso}}(n_1) = 40$. En utilisant le temps minimum, nous stockons $d_{\text{head}(n_1)-n_1} = 2$ et en-ligne nous calculons $\mathit{RWCET}_{\mathit{iso}}(f_{0,1}) = 50$, $\mathit{RWCET}_{\mathit{iso}}(n_1) = 48$, $\mathit{RWCET}_{\mathit{iso}}(f_{1,1}) = 45$, $\mathit{RWCET}_{\mathit{iso}}(n_1) = 43$.*

Définition 29 ($\mathsf{W_{max}}$)**.** *On applique également l'analyse de WCET restant afin de déterminer W_{max} le pire temps entre deux points d'observation successifs dans le scénario charge maximale.*

$$W_{max} = max_{x,x'}(\mathit{RWCET}_{max}(x) - \mathit{RWCET}_{max}(x'))$$

Lorsque le contrôleur est logiciel, le calcul des WCET restant doit prendre en compte les instrumentations (Def. 18). Ainsi, en charge maximale, C_R et F_D sont toujours exécutés entre deux points. En *isolation* ou *critiques uniquement*, C_R est toujours évalué entre deux points et N_E est exécuté à la fin.

5.4 Phase en-ligne

Dynamiquement, le contrôleur surveille le temps d'exécution en cours, calcule le WCET restant et évalue la condition de sûreté. Si besoin, il interrompt les tâches moins critiques.

5.4.1 Calcul dynamique de $\mathsf{RWCET}_{\mathsf{iso/cri}}(x)$

L'idée est de calculer une borne supérieure de $\mathsf{RWCET}_{\mathsf{iso/cri}}(x)$ à partir des informations stockées avec une complexité linéaire. Le calcul en-ligne de $\mathsf{RWCET}_{\mathsf{iso/cri}}(x)$ est décrit dans l'algorithme 1. La preuve de correction est donnée dans [KBP14]. Les données pré-calculées sont stockées en mémoire en tant que tableaux constants : *level* pour le niveau, *d* et *w* pour les WCETs partiels, *type* pour le type. L'algorithme maintient deux variables locales *o_level* (pour le dernier niveau observé) et *ll* pour le niveau local. Comme les appels de fonction relient les ECFGs, nous devons déterminer en-ligne si il y a un passage d'un ECFG à un autre, ce qui est possible grâce au $type(x)$ de chaque point. La variable *offset* calcule le niveau d'imbrication cumulé jusqu'au dernier appel de fonction. Le niveau local *ll* dépend à la fois de cet *offset* et du niveau d'imbrication statique du point d'observation courant. L'algorithme stocke également deux tableaux : *last_point* du dernier point observé et R contenant le $\mathsf{RWCET}_{\mathsf{iso/cri}}$ calculé par niveau.

Algorithme 1: Calcul dynamique de $\mathsf{RWCET}_{\mathsf{iso/cri}}(x)$

```
Pre-computed data : level, w, d, type
Input : x
Data : o_level = 0, ll = level[x], last_point[0]=start, R[0]=WCET_iso/cri, offset = 0
Output : RWCET_iso/cri(x) = R[ll]
if (type[x] ==F_EX or F_NX) then                          /* condition 1 */
    o_level = o_level − 1
    offset = offset − level[x]
ll = offset + level[x]
if o_level < ll then                                      /* condition 2 */
    R[ll] = R[ll − 1] − d[x]
else
    if (last_point[ll] == x) then                         /* condition 3 */
        R[ll] = R[ll] − w[x]
    else
        R[ll] = R[ll − 1] − d[x]
last_point[ll] = x
o_level = ll
if (level[x] ==F_EN or F_NX) then                         /* condition 4 */
    offset += level[x]
```

Cinq cas de figure se présentent durant l'exécution :

- Cas 1 : Lorsqu'on entre dans une fonction (*condition 4*), on incrémente l'offset par le niveau du point d'entrée.
- Cas 2 : le niveau courant *o_level* est moins élevé que le niveau local *ll*. Dans ce cas, l'ECFG est traversé en direction « avant », c'est-à-dire qu'on entre dans une boucle. Le WCET restant vaut le WCET restant au point supérieur *c* moins le temps entre *c* et le point d'observation x, $d[x] = d_{c-x}$.
- Cas 3 : $o_level \geq ll$ et $last_point[ll] = x$. Cela se produit lorsque x est revisité et donc que l'ECFG est traversé en direction « arrière ». Le temps restant à ce niveau est alors réduit de $w[x] = w_c$.
- Cas 4 : lorsqu'on sort d'une fonction (*condition 1*), on décrémente l'offset du niveau du point d'entrée. De plus, *o_level* est décrémenté de 1 pour indiquer qu'on a traversé un point de sortie.
- Cas 5 : sinon. L'ECFG est traversé en direction « avant » et en séquence par rapport au dernier point observé (séquence ou if-then-else). Le temps restant est alors le temps restant au point supérieur moins $d[x] = d_{c-x}$.

De plus, d'après la définition 24, le niveau d'un point de sortie est égal au niveau du point d'entrée. Ainsi, lorsque l'offset est nul, on se trouve dans la fonction principale F_0.

Exemple 33. *Un exemple de calcul de* $\mathsf{RWCET}_{\mathsf{iso}}(x)$ *de la figure 5.5(a) est donné dans la table 5.2.*

5.4.2 Implantation logicielle

Nous avons porté le contrôleur logiciel sur le Tms en respectant le modèle d'exécution 3. Pour cela, nous avons réutilisé : la configuration mémoire de la figure 4.3(b), la librairie bare-metal décrite dans la partie 4.3, la barrière de synchronisation, les compteurs TCSL et TCSH pour estimer en chaque point le temps d'exécution déjà écoulé.

Obs. Point	*condition*				Offset	$RWCET_{iso}$(x)	Last point[ll]	Obs. level
	1	2	3	4				
start					0	$R[0] = RWCET_{iso}$	LP[0]=*start*	0
$n_{0,a}$	0	1		0	0	$R[1] = R[0] - 0$	LP[1]=$n_{0,a}$	1
$f_{0,1}$	0	0	0	1	0	$R[1] = R[0] - d_{start-\mathrm{f_{0,1}}}$	LP[1]=$f_{0,1}$	1
$n_{1,a}$	0	1		0	1	$R[2] = R[1] - d_{\mathrm{f_{0,1}-n_{1,a}}}$	LP[2]=$n_{1,a}$	2
c	0	0	0	0	1	$R[2] = R[1] - d_{\mathrm{f_{0,1}-c}}$	LP[2]=c	2
$n_{1,b}$	0	1		0	1	$R[3] = R[2] - d_{\mathrm{c-n_{1,b}}}$	LP[3]=$n_{1,b}$	3
c	0	0	1	0	1	$R[2] = R[2] - w_{\mathrm{c}}$	LP[2]=c	2
$n_{1,b}$	0	1		0	1	$R[3] = R[2] - d_{\mathrm{c-n_{1,b}}}$	LP[3]=$n_{1,b}$	3
c	0	0	1	0	1	$R[2] = R[2] - w_{\mathrm{c}}$	LP[2]=c	2
$n_{0,b}$	1	0	0	0	0	$R[1] = R[0] - d_{start-\mathrm{n_{0,b}}}$	LP[1]=$n_{0,b}$	1

Table 5.2 – Exemple de calcul en ligne de $\mathsf{RWCET_{iso/cri}}$

La suspension et le redémarrage des tâches moins critiques sont implantés en utilisant les mécanismes d'événements et d'interruption du Tms. La librairie bare-metal a donc été étendue avec des fonctions permettant de (1) configurer les événements et les interruptions du Tms, (2) utiliser des événements et (3) suspendre ou reprendre les tâches moins critiques. Nous explorons le comportement de notre méthodologie en jouant sur les facteurs suivants :

1. la taille de l'application critique ;
2. le nombre de tâches critiques, i.e de 1 à 2 ;
3. le nombre de cœurs exécutant des tâches moins critiques, i.e de 1 à 7 ;
4. l'échéance D_C d'une tâche critique, i.e. de valeurs très proches de $WCET_{iso/cri}$ jusqu'à des valeurs beaucoup plus relâchées ;
5. la granularité des points d'observation : HP1 (jusqu'à *level* =1), HP2 (jusqu'à *level* =2) et HP3 (jusqu'à *level* =3).

Comme il n'existe aucun outil d'analyse statique pour le Tms, nous avons utilisé des techniques de mesures similaires à celles de la partie 3.5 pour évaluer (1) les informations structurelles et temporelles ($d(x)$, $w(x)$ et $\mathsf{W_{max}}$) ; (2) les performances du contrôleur logiciel.

Etude de cas

Nous sommes partis du programme *gemm* de la suite de cas d'étude Polybench [Pou13], décrit dans l'algorithme 2. Toutes les fonctions critiques sont donc des fonctions *gemm*. Les tâches moins critiques

Algorithme 2: Cas d'étude

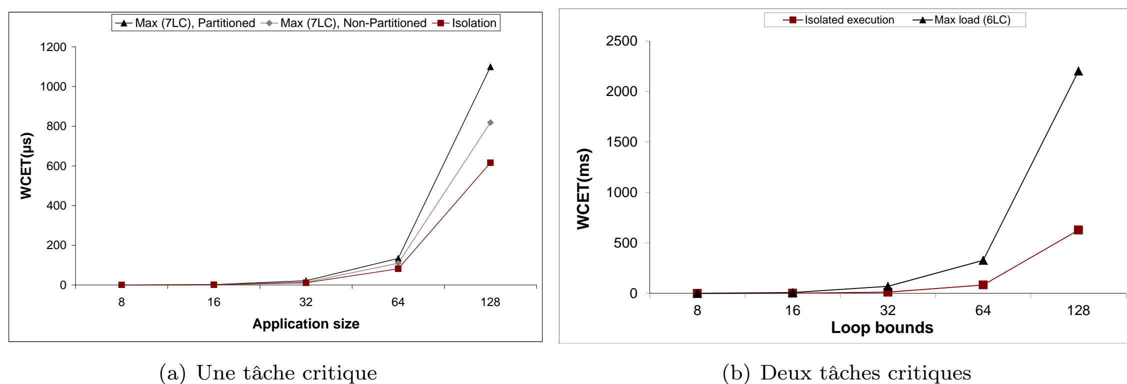
```
# define PB_N 100
int A[PB_N][PB_N];
int B[PB_N][PB_N];
int C[PB_N][PB_N];
int alpha=32412;
int beta=2123;
begin int main()
    gemm();
    return EXIT_SUC;

begin int gemm()
    int i, j, k;
    for (i = 0; i < PB_N; i++) do
        for (j = 0; j < PB_N; j++) do
            C[i][j] *= beta;
            for (k = 0; k < PB_N; k++) do
                C[i][j] += alpha * A[i][k] * B[k][j];
    return EXIT_SUCCESS;
```

génèrent beaucoup d'accès à la DDR, et donc d'interférences, car leurs données sont stockées dans la mémoire globale et sont non cacheables (puisque le L2 est en SRAM).

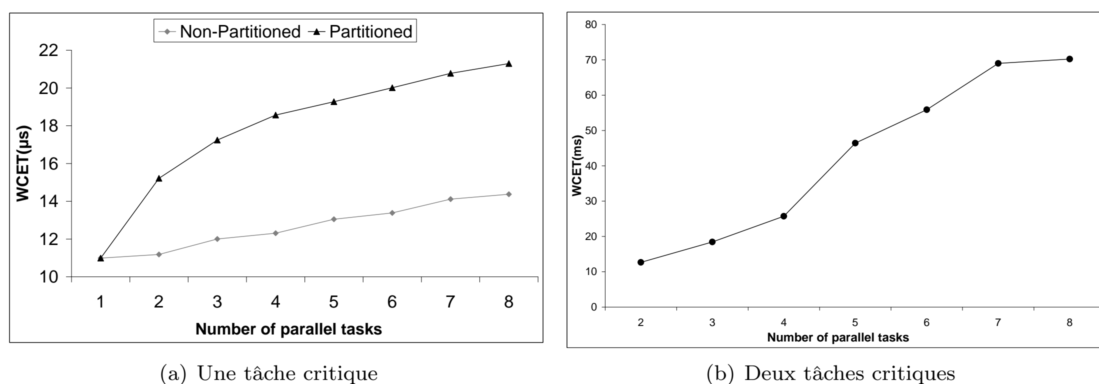
On évalue les différents WCET dans la figure 5.8 en fonction de la taille de l'application, i.e. $N \in \{8, 16, 32, 64, 128\}$, soit en présence d'une tâche critique seule (fig. 5.8(a)) ou de deux tâches critiques en mode partitionné (fig. 5.8(b)). La mémoire DDR peut être partitionnée ou non. On observe que la différence s'accroît lorsque le nombre de congestions augmente, ce qui est en corrélation avec l'idée de la méthode.

La figure 5.9(a) montre la variation du WCET en charge maximale pour des tâches concurrentes allant de 0 à 7 pour une taille fixe (N=32). La figure 5.9(b) présente les mêmes résultats pour une taille de 128



(a) Une tâche critique (b) Deux tâches critiques

FIGURE 5.8 – Variation des WCETs en fonction de la taille



(a) Une tâche critique (b) Deux tâches critiques

FIGURE 5.9 – Variation des WCETs en fonction du nombre de tâche en parallèle

en mode partitionné. Dans les deux cas, lorsqu'il n'y a pas de tâche concurrente, on retrouve $\mathsf{WCET_{iso}}$ (resp. $\mathsf{WCET_{cri}}$). On observe que :

— le WCET varie en fonction du nombre de tâches en parallèle. Il y a notamment une augmentation d'un facteur 3 en mode partitionné dans le cas d'une tâche. La variation est encore plus importante quand il y a deux tâches de plus grande taille ;

— le WCET en mode partitionné est celui requis en avionique et produit des temps d'exécution plus lents à cause du coût d'accès dans des zones diverses.

De plus, nous avons évalué le coût du contrôleur logiciel :

	Read timer	F_D quand C_D=false	Suspend/Restart
Max.time (Cycles)	70	501	1966

Résultats

Pour évaluer les performances du contrôleur, nous utilisons deux métriques :

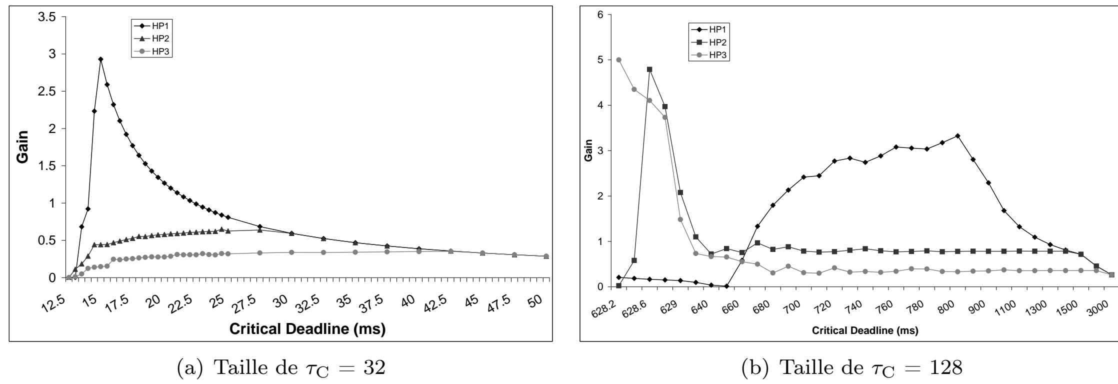
Définition 30 (Gain relatif)**.** *Soit t le temps d'exécution d'une tâche moins critique avec notre méthode et $t_{iso/cri}$ le temps d'exécution en isolation (resp.* critiques uniquement*). Le* gain relatif *obtenu est :*

$$(t - t_{iso/cri})/t_{iso/cri}$$

Définition 31 (Gain absolu)**.** *Soit RT le temps de réponse de la tâche moins critique avec notre méthode et $RT_{iso/cri}$ le temps de réponse en isolation. Le gain absolu est :*

$$RT_{iso/cri} \text{ - } RT$$

Les figures 5.10(a) et 5.10(b) présentent les gains relatifs obtenus pour plusieurs configurations de points d'observation et deux tailles d'application en présence de deux tâches critiques. Nos observations sont les suivantes :



(a) Taille de $\tau_C = 32$ (b) Taille de $\tau_C = 128$

FIGURE 5.10 – Gain relatif pour plusieurs échéances et plusieurs niveaux

- Plus l'échéance est serrée, plus le contrôleur passe rapidement en isolation (resp. *critiques uniquement*). Dans ce cas, il vaut mieux utiliser des points d'observation précis ;
- Lorsque l'échéance est intermédiaire, HP1 offre de bons résultats car on observe moins souvent le comportement et on change peu de mode ;
- Lorsque l'échéance est très lâche, le contrôleur devient inutile et rajoute même un sur-coût inutile.

5.5 Conclusion

Les résultats obtenus à l'implantation prouvent à la fois la faisabilité de l'approche et la viabilité car les gains observés sur le cas d'étude sont très prometteurs. Ces travaux restent néanmoins très préliminaires et nous devons (1) continuer nos mesures sur davantage d'applications afin d'identifier celles pour lesquelles notre méthode est bénéfique ; (2) améliorer les performances en activant moins souvent le contrôleur ou en optimisant la position des points ; (3) étendre les niveaux de criticité ; (4) améliorer les stratégies d'adaptation. Les deux derniers points seront étudiés prochainement dans le projet DREAMS.

Chapitre 6

Bilan et perspectives

6.1 Conclusion

6.1.1 Bilan sur la programmation sûre de plates-formes embarquées de type multi/pluri-cœurs

Mes travaux d'encadrement et de recherche ont abouti à la mise en place d'un environnement de développement de bout en bout allant de la spécification formelle d'un assemblage multi-périodique jusqu'à l'exécution prédictible sur des cibles multi/pluri-cœurs.

Langage Prelude

Le langage Prelude permet de spécifier formellement des systèmes multi-périodiques en assemblant des blocs élémentaires écrits dans d'autres langages de programmation. Il répond aux besoins d'expressivité, de validation et d'analyse, notamment des systèmes de contrôle/commande, cela a été illustré au travers de nombreux exemples.

Des collaborations récentes avec des collègues en automatique nous ont prouvé qu'il facilitait les discussions en phase d'intégration, car il est simple à comprendre pour les deux mondes et permet de faire ressortir le lien entre certains choix de la spécification de haut niveau et les conséquences sur le bas niveau temps réel. Le maintien d'une vue commune pendant toute la phase de développement offre la possibilité de vérifier régulièrement les exigences exprimées en amont. Ce besoin de développement en coopération ne va que croître avec l'arrivée de contrôleurs plus réactifs du fait des structures composites plus souples, du besoin de réduction de consommation et de l'intensification du trafic aérien.

Certaines contraintes de déterminisme imposées par Prelude étant encore trop proches d'un monde de programmation, les extensions proposées par Rémy Wyss se rapprochent encore davantage des méthodes et habitudes de spécification des automaticiens en acceptant des libertés dans la consommation des données. Le travail à fournir est donc plus aisé et les outils de recherche automatique aident ensuite à trouver des implantations respectant les contraintes locales.

Le nombre de nœuds à assembler peut être un frein à l'utilisation de Prelude. Nous travaillons avec Thomas Loquen (ONERA - DCSD) sur la génération de programmes Prelude à partir de modèles Matlab/Simulink. Une autre piste de simplification explorée avec Luca Santinelli [LB13 ; SPB14] est la recherche de stratégies de regroupement pour ne plus manipuler qu'un nombre restreint de nœuds. Antoine Bertout et Julien Forget [BF14 ; BFO14] étudient également des techniques de regroupement.

Embarquement prédictible de cibles multi/pluri-cœurs

L'utilisation des technologies multi/pluri-cœurs va devenir incontournable dans les années à venir. Il faut donc trouver des solutions génériques assurant leur prédictibilité. La définition de règles d'utilisation promue par les modèles d'exécution est une première étape dans ce sens. Ce qu'il ressort de nos différentes solutions est qu'il faut tendre vers des architectures distribuées dont on sait maîtriser les comportements depuis de nombreuses années (comme les architectures fédérées ou IMA). La meilleure solution serait que les puces intègrent directement les mécanismes explicites qui nous intéressent. La combinaison d'une partie de ces mécanismes matériels prédictibles (comme ceux offerts par le Tms ou le Tilera) et de

règles de bon codage peut néanmoins aboutir à un résultat valide même si moins performant, comme nous l'avons montré sur plusieurs études de cas.

Nos expérimentations sur la partie outillage et génération de placement sont prometteuses car nous n'avons pas rencontré de réels problèmes de faisabilité. Il faut néanmoins poursuivre l'effort pour automatiser certaines étapes comme : extraction automatique des entrées depuis les programmes PRELUDE, description de la cible à partir d'une exploration, génération automatique des fichiers de configuration et code pour la cible.

Un des points faibles de notre solution reste son manque de généricité, le coût pour investiguer une nouvelle plate-forme et re-développer les exécutifs n'étant pas encore acceptable.

Conservation de la prédictibilité en présence d'applications moins critiques

Plus récemment, nous avons mélangé sur un multi-cœurs respectant un modèle d'exécution spécifique des applications critiques avec des moins critiques. Notre choix est de garantir coûte que coûte le respect des échéances temporelles des applications critiques tandis que nous n'assurons aucun contrat pour les autres. Un contrôleur en-ligne régule le comportement des tâches moins critiques pour qu'elles n'interfèrent pas dangereusement avec les critiques tout en maximisant leur utilisation CPU. Ce travail en cours sera poursuivi avec mes collègues toulousains et dans le projet européen DREAMS.

6.1.2 Thématiques non décrites dans le document

Certains des travaux auxquels j'ai collaboré n'ont pas été présentés dans le document et j'en donne un aperçu très bref dans la suite.

Vérification de propriétés temps réel de bout en bout sur plates-formes avioniques IMA

Le document n'a décrit que des activités autour de la programmation d'un composant unique mais un calculateur est toujours intégré dans une architecture avionique plus large. Les travaux de thèse de Michaël Lauer (co-encadrée avec Frédéric Boniol et Jérôme Ermont ; financée par le projet ANR Satrimmap 2009-2012) ont permis de vérifier des exigences temps réel sur une plate-forme IMA. Ces exigences s'expriment sur des chaînes fonctionnelles, c'est-à-dire des séquences de fonctions non nécessairement exécutées sur un même calculateur (version distribuée des concepts de la partie 2.5.1), et spécifient une borne acceptable (minimale ou maximale) pour une propriété temporelle. Nous avons identifié trois catégories d'exigences temps réel : latence, fraîcheur et cohérence. De part la nature d'une architecture IMA, la vérification de propriétés temps réel doit tenir compte de l'ordonnancement réalisé par le système d'exploitation et la gestion des paquets dans un réseau de type AFDX. Les systèmes IMA sont décrits dans le formalisme du *Tagged Signal Model* [LS96] et chaque exigence est vérifiée en analysant un programme linéaire mixte, c'est-à-dire contenant à la fois des variables entières et réelles. Ces travaux ont donné lieu à plusieurs publications [LEPB10 ; LEBP11a ; LEBP11b ; LBEP11 ; BPLE12 ; BLPE13 ; LBPE14].

Guillaume Brau, durant son master, a étudié les mêmes propriétés de latence pour des architectures mixant des réseaux de type TTEthernet et des ordonnanceurs à priorité fixe. Son travail, publié dans un junior workshop [BP12], a consisté à simuler de tels systèmes avec l'outil PTOLEMY II. Les résultats obtenus ne sont pas des bornes supérieures mais des distributions de valeurs atteintes.

Utilisation de plates-formes avioniques IMA nouvelles générations

J'ai également contribué dans le projet européen FP7 SCARLETT (2008-2012) à la définition de plate-forme avionique IMA-2G reconfigurable. La première étape a consisté à définir la notion de reconfiguration : l'objectif était d'améliorer la fiabilité opérationnelle et donc de réduire les coûts dus aux annulations ou retards de vols. En aucun cas, la reconfiguration ne visait à améliorer la fiabilité, mais elle ne devait bien évidemment pas la dégrader. En collaboration avec Airbus et Eric Noulard, nous avons défini le système de reconfiguration dont le fonctionnement est le suivant : détection d'une panne d'un module, décision par un nouvel élément de la plate-forme appelé le *superviseur de reconfiguration* de vérifier la panne, d'arrêter le module défectueux, de choisir une reconfiguration en analysant un graphe de configuration prédéfini, de relocaliser les fonctions du module défectueux sur un module de réserve, puis de vérifier globalement la cohérence de la plate-forme. Le deuxième travail (en collaboration avec Pierre Bieber et Julien Brunel) a consisté à réaliser un certain nombre d'analyses de sûreté de fonctionnement : FHA détaillée du système de reconfiguration, analyse préliminaire de safety (PSSA) en utilisant le langage

AltaRica et l'outil Cecilia OCAS, génération de scénarios de test pour le démonstrateur avionique réalisé par Airbus. Ces travaux ont donné lieu à plusieurs papiers [BBB12 ; PBB12 ; BNB10 ; BNP09].

Coopération d'outils de simulation

J'ai également co-encadré le post-doctorat de Gilles Lasnier avec Janette Cardoso et Pierre Siron (ISAE) sur l'aide à la conception de systèmes cyber-physiques à base de simulation. Certaines analyses de l'exécution des applications nécessitent de connaître précisément les réactions sur l'architecture cible et arrivent tardivement dans le processus de développement. L'objectif du post-doctorat était de proposer des méthodes et outils permettant d'analyser le plus tôt possible les comportements (même partiels) futurs sur la cible, en tenant compte par exemple de l'interaction avec des capteurs / actionneurs, ou encore du placement sur la plate-forme distribuée. L'approche suivie pour étudier le comportement du système complet est de réaliser des simulations intégrant plusieurs types de modèles réalisés à différents niveaux dans la phase de conception.

Une telle activité repose sur la connaissance de plusieurs domaines transverses (logiciel ou matériel) et nécessite l'utilisation de modèles hétérogènes (tels que continus ou discrets). Nous avons développé un environnement de co-simulation permettant la coopération de deux outils de simulation open source : Ptolemy II [Pto14] et HLA/CERTI [IEE10 ; NRS09]. Nous avons vu que Matlab/Simulink était largement répandu pour le développement d'applications de contrôle/commande. Ptolemy II [Pto14] est un environnement open source de modélisation et de simulation, équivalent à un sous-ensemble de Simulink, développé à l'université de Californie à Berkeley. Un de ces atouts, en plus d'être open source, est de permettre la cohabitation de plusieurs modèles de calcul (models of computation MoC). Le standard de simulation HLA est dédié aux simulations distribuées interopérables (parfois en intégrant des éléments matériels) et est davantage utilisé en phase de conception détaillée ou d'intégration.

La coopération de ces deux environnements permet de combiner les avantages de chacun : hétérogénéité fournie par Ptolemy II (possibilité d'utiliser plusieurs modèles de calcul) et interopérabilité offerte par HLA (possibilité d'utiliser plusieurs modèles de simulation, des codes et des équipements physiques). Ces travaux ont donné lieu à plusieurs publications [LCS13 ; LCPS13]. L'étude de cas Rosace a été recodée en Ptolemy II et simulée sur la plate-forme PRISE [CCH14] en interfaçant le contrôleur avec des composants HLA déjà existants (capteurs et actionneurs) et un manche donnant les ordres de changement de niveaux de vol.

6.2 Projet de recherche

Cette partie est dédiée à la description de pistes de recherche pour les prochaines années découlant des activités menées jusque là. Le document a essentiellement présenté des résultats sur la maîtrise du temps réel, et notamment de la prédictibilité. Ce n'est qu'un point de vue de l'embarquement, et dans les prochaines années je souhaite traiter les aspects temps réel et tolérance aux fautes en même temps. En effet, la prise en compte de la sûreté de fonctionnement dès les phases amont et de conserve avec les méthodes, outils et exécutifs de développement est tout aussi fondamentale. En résumé, la thématique générale est la contribution au développement sûr de systèmes embarqués critiques temps réel et tolérants aux fautes. Le projet est structuré en trois grandes directions :

1. méthodologie : de façon à s'insérer dans les processus de développement industriels et préparer la phase de certification ;
2. langage, compilation et génération de configuration sur les cibles : de façon à offrir des outils à l'intégrateur pour programmer et configurer ces plates-formes en assurant la correction temporelle et sûreté ;
3. architectures et exécutifs : de façon à offrir des supports exécutifs prédictibles et fiables sur les architectures embarquées actuelles et à venir.

6.2.1 Vers une méthodologie de développement de bout en bout

Le premier axe de recherche concerne l'intégration de solutions d'automatisation ou de vérification ou de validation dans les processus de développement industriels ou la proposition d'adaptation de ces processus en fonction des nouveautés.

Automatisation de la génération de code à partir de spécifications de haut niveau La chaîne d'outils allant d'un programme PRELUDE vers un code bare-metal est à peu près claire même si certaines phases doivent encore être automatisées comme mentionné précédemment. L'étape de transition entre l'automatique et l'informatique reste encore à approfondir. Nous travaillons dans le projet interne ONERA FORCES 3 (porteur Virginie Wiels, contributeurs DTIM et DCSD, 2013-2015), à la génération de code prouvé correct pour des contrôleurs de vol de drone. Une partie des travaux concerne la translation de modèles exprimées en MATLAB/SIMULINK vers nos langages et outils tels que LUSTRE pour les blocs élémentaires et PRELUDE pour la partie assemblage. Ces transformations repartent des travaux de [TSCC05] et la branche GAL du projet GENEAUTO `http://geneauto.gforge.enseeiht.fr`.

Un des points clés pour assurer la correction est la traçabilité des exigences du monde automatique durant les phases de développement. Les ingénieurs en automatique considèrent : (1) des propriétés de stabilité, robustesse et performance ; (2) des validations et vérifications en boucles fermées. Dans l'étude de cas ROSACE, nous avons montré la faisabilité de maintenir les propriétés de performances et les avons analysées à base de simulation, en implantant une version discrétisée de l'environnement. Nous pourrions encore parfaire notre méthodologie en :

— remplaçant la simulation par une méthode formelle ou un calcul analytique ;
— étendant les propriétés traitées ;
— fermant la boucle autrement qu'en portant le modèle discrétisé de l'environnement sur un cœur.

Notre méthodologie trace les exigences de haut niveau jusqu'à l'exécution réelle, cependant nous pourrions également fournir des outils et méthodes aux automaticiens pour qu'ils prennent en compte en amont le comportement temporel de leur contrôleur (WCET ou ordonnançabilité ou délai). Aujourd'hui, la conception se base essentiellement sur des bonnes pratiques telles que minimiser la gigue, prendre en compte les délais réseau entre capteurs/actionneurs et calculateurs. Nous pourrions améliorer leurs connaissances sur la dimension temporelle et la coopération de simulation proposée dans le post-doctorat de Gilles Lasnier est une première idée pour remonter des informations au niveau spécification.

Intégration dans des approches multi-vues La conception de systèmes complexes repose également sur l'utilisation d'approches à base de modèles de façon à avoir une vue globale du système tenant compte des différentes *facettes métier* (fonctionnel, temps réel, sûreté de fonctionnement, compatibilité électromagnétique, performances). Les langages de description de systèmes comme SYSML [OMG10] ou d'architectures comme AADL [FGH06] offrent la possibilité de spécifier de telles vues globales. En AADL, plusieurs travaux prennent en compte les aspects bas niveau temps réel dans AADL [RDS14 ; CRB13] et/ou la sûreté de fonctionnement [RFKK08]. Gérer la prise en compte de plusieurs vues s'accompagne d'un grand nombre de difficultés telles qu' (1) assurer la cohérence des vues, (2) offrir des outils pour focaliser sur une vue afin d'appliquer des méthodes spécifiques et revenir dans la vue globale mise à jour, (3) valider les raffinages. Il serait intéressant d'insérer nos méthodes d'intégration d'applications multi-périodiques sur des plates-formes embarquées dans une approche multi-vues.

Certification incrémentale Enfin, un impératif dans l'industrie aéronautique est de fournir des méthodologies génériques aux développeurs pour certifier et valider leurs applications tout en minimisant les coûts. La réponse apportée jusque là est la *certification incrémentale* : une application est étudiée en isolation sur la cible (*stand-alone*) et les comportements lorsque les autres se partagent le processeur sont extrapolés. L'embarquement des architectures multi/pluri-cœurs doit s'accompagner de nouvelles solutions de certification : (1) l'absence de composabilité temporelle nécessite de réfléchir à de nouvelles stratégies en utilisant des composants génériques ou en bornant les effets des autres applications ; (2) l'introduction de davantage d'adaptativité requiert de nouvelles méthodes pour certifier des familles de configurations ; (3) la complexification des systèmes entraînera des besoins d'analyse, de vérification et validation du système dans son ensemble.

6.2.2 Langage, compilation et génération de configurations

Le deuxième axe de recherche concerne les extensions possibles du langage PRELUDE ainsi que l'outillage associé.

Langage temps réel tolérant aux fautes La première extension concerne la prise en compte de la tolérance aux fautes. Les multi/pluri-cœurs sont soumis à deux types de fautes [Bor05] : (1) fautes permanentes en raison de phénomènes physiques divers comme le vieillissement, la fatigue ou la mortalité

infantile des composants. Dans ce cas, le composant est définitivement endommagé ; (2) transientes à cause d'événements extérieurs tels qu'une particule de haute énergie frappant le circuit (ex. SEU - Single Event Upset) et altérant temporairement des données. Le concepteur doit démontrer, entre autres, que l'exécution de l'application est robuste aux fautes issues du matériel. Ces fautes doivent donc être prises en compte à la conception de façon à offrir des moyens pour résister à un certain nombre d'elles, on parle alors de tolérance aux fautes. A titre d'exemple, si l'on embarque un multi/pluri-cœurs composé de 100 cœurs tel que le taux d'arrêt temporaire d'un cœur (on suppose qu'un cœur subissant un arrêt temporel fonctionne correctement après un reboot de la puce) est $10^{-3}/FH$ pour un vol durant 10h, alors on sait que durant le vol on perdra en moyenne un cœur. Il faut prévoir des moyens de tolérer cette erreur en vol tout en maintenant les fonctionnalités.

Les fautes permanentes sont encore peu étudiées dans la littérature car les technologies multi/pluri-cœurs sont récentes et il faut établir formellement les modèles de faute des composants internes (cœurs, routeurs ou mémoire). Les fautes transientes sont davantage analysées, notamment dans le cas d'un processeur simple et leurs modèles doivent être étendus pour les multi/pluri-cœurs. A partir de ces modèles, il faudra proposer des mécanismes de tolérance aux fautes tels que des reconfigurations dynamiques et sûres.

Kevin Delmas débute une thèse (co-encadrée avec Rémi Delmas) à partir de novembre 2014 dont l'objectif est de développer des méthodes et outils génériques permettant la programmation temps réel sur multi/pluri-cœurs, et exploitant la multiplicité des unités de calcul disponibles pour assurer la redondance des calculs nécessaires afin de tolérer les fautes de type SEU. L'objectif est donc (1) d'étendre le langage Prelude de façon à prendre en compte la tolérance aux fautes ; (2) de proposer un processus automatique de durcissement de l'architecture afin de respecter le niveau de sûreté requis en intégrant des redondances et des mécanismes de sûreté ; (3) de générer de scénarios de test pour injection de fautes sur la cible. Kevin a commencé à regarder le deuxième point durant son stage de master : à partir d'une architecture nominale, le processus de durcissement sélectionne un sous-ensemble de composants qui sont remplacés par des *patrons de sûreté* [KSB04 ; RFKK08]. Ce résultat préliminaire est publié sous forme d'un papier court [DDP14].

Langages parallèles temps réel Une deuxième extension concerne la parallélisation automatique d'applications. La démarche suivie avec l'environnement Prelude-SchedMCore est de laisser l'utilisateur découper son application en un ensemble de nœuds à assembler et d'évaluer les possibilités de parallélisation à ce niveau. Un réel effort doit être fourni pour offrir un ou des langages de programmation suffisamment abstraits et simples ainsi qu'un ensemble d'étapes de transformation permettant de compiler de tels programmes jusqu'au niveau parallèle et temps réel. Peu de travaux existent dans ce domaine et nous pouvons mentionner le projet P-Socrates [PQB14] dont l'objectif est la parallélisation d'applications temps réel en utilisant des pragmas dans l'idée d'OpenMP.

Génération de configuration Une troisième question est le choix de la méthode et de l'outil de placement. La plupart de mes travaux reposent sur l'utilisation d'OPL mais il existe d'autres solutions, notamment en utilisant une approche SMT (Satisfiability Modulo Theories) dont les outils sont extrêmement performants. Nous commençons un projet interne ONERA, appelé MAUSART (MApping Under SAfety and Real-Time constraints - porteur Rémi Delmas, contributeurs DTIM et DCSD, 2015-2017), dont l'objectif est de comparer et d'interfacer plusieurs outils de recherche de placement. De la sorte, l'utilisateur pourra récupérer les résultats si l'un au moins des outils parvient à trouver une solution et ce sans recoder son problème dans plusieurs langages. Un autre point abordé dans le projet est la question du rebond : comment, à partir d'un placement déjà calculé et l'évolution de la spécification, regénérer un nouveau placement conservant un certain nombre de propriétés des placements précédents.

6.2.3 Architectures et exécutifs temps réel tolérants aux fautes

Le troisième axe de recherche concerne la programmation des plates-formes embarquées.

Généricité de la programmation sur un multi/pluri-cœurs Un obstacle, déjà évoqué plusieurs fois, est le manque de généricité de nos modèles d'exécution et d'exécutifs bare-metal, lequel est lié à la grande diversité d'architectures multi/pluri-cœurs. Afin de remédier, au moins partiellement, à ce problème, nous devons généraliser nos méthodes afin de pouvoir porter rapidement nos applications sur une cible quelconque (ou montrer que la cible n'est pas adaptée).

Pour ce faire, une première étape est la définition d'une méthodologie d'exploration d'architecture. Les plus mauvaises configurations étant difficiles à identifier par manque d'informations précises sur le matériel, le but est d'automatiser la recherche des scénarios pire cas. Les *stressing benchmarks* [NP12 ; BGG14 ; Bin14] sont un premier pas en ce sens. Ensuite, les comportements étant complexes à analyser du fait d'une grande combinatoire, il faut également déduire, de la phase d'exploration, des patrons d'utilisation prédictibles et un modèle micro-architectural nécessaire aux outils de calcul statique de WCET. Des travaux récents reposent sur l'utilisation de méthodes formelles et d'une vision plus haut niveau pour générer des drivers adaptés [TBB13].

Une deuxième étape est le développement de techniques de codage et d'organisation d'architectures logicielles des librairies bare-metal de façon à minimiser les efforts de redéploiement. L'instrumentation automatique et non intrusive de l'exécutable sur la cible doit également être davantage examinée.

Gestion de la criticité mixte sur un multi/pluri-cœurs Le dernier chapitre a ouvert la discussion sur le partage d'un multi-cœurs entre plusieurs applications avioniques. L'extension des principes de l'Arinc 653 [Aer97] aux multi/pluri-cœurs est un besoin réel et il faut imaginer des solutions pour implanter les notions de ségrégation et criticité mixte sur ces plates-formes. Pour atteindre un tel résultat, plusieurs verrous doivent être levés.

D'un point de vue temporel, il faut assurer la prédictibilité des tâches critiques mais également garantir des contrats de qualité de service pour les moins critiques. Une première étape est donc la définition de la notion de contrat pour les tâches, par exemple sous la forme d'un taux d'accès moyen sur le bus partagé ou un nombre minimal de requêtes à la mémoire. La deuxième tâche sera de proposer des mécanismes pour maintenir les différentes qualités de service requises.

Concernant la sûreté de fonctionnement, un point important est la ségrégation, c'est-à-dire le confinement des comportements des applications pour ne pas impacter voire modifier les exécutions des autres. Il est nécessaire d'inventer des mécanismes de partitionnement garantissant un partage sûr du calculateur entre plusieurs applications. [CRS11] Un point clé pour ce faire sera d'étudier des méthodes de gestion de la mémoire [YMWP14] et de routage. Un deuxième objectif est le confinement des erreurs de chaque application pour ne pas corrompre les autres.

Nous allons aborder une partie de ces problématiques dans un projet interne, appelé Macsima (ManyCore System for Integrated Modular Avionics - porteur Eric Noulard, contributeurs DCSD et DEMR, 2015-2018). Le démonstrateur mélangera trois applications avioniques : (1) traitement/fusion des données en sortie des différents capteurs (type traitement du signal), (2) gestion de la mission de l'aéronef (sa planification ou sa replanification en fonction des événements, son optimisation, etc.), (3) guidage et pilotage de l'aéronef, son maintien en état de fonctionnement

Infrastructure et middelware temps réel à criticité mixte tolérant aux fautes sur une plate-forme embarquée Les plates-formes embarquées pour les futurs systèmes critiques aéronautiques ou spatiaux vont encore évoluer. Une telle architecture candidate pourrait être qualifiée par le terme « *mega-pluri-cœurs* » dans le sens où quelques pluri-cœurs (de l'ordre 5 à 10, pour des besoins de tolérance aux fautes) connectés par un réseau temps réel à haut débit exécuteraient parallèlement plusieurs applications de criticité potentiellement différentes. Les avantages escomptés sont nombreux, en particulier (1) une réduction globale de la consommation électrique, du poids, et de l'encombrement, (2) une capacité accrue d'un ordre de grandeur, par exemple multipliée par 20 pour 5 calculateurs contenant 1024 cœurs chacun par rapport à l'avionique de l'A350, et (3) une forte centralisation des traitements, permettant le développement de systèmes plus réactifs et plus synchrones (par exemple entre un traitement radar, l'interprétation des pistes détectées, et la gestion de la mission).

Un effort de virtualisation doit être réalisé pour les concepteurs pour leur cacher la complexité bas niveau. Les solutions actuelles doivent donc être étendues voire repensées pour définir des stratégies globales de gestion exécutive – en particulier du point de vue des deux niveaux de communications (réseau et NoC) et du partitionnement –, de déploiement et développer des outils d'analyse associés.

Chapitre 7

CV détaillé

7.1 Curriculum vitae

Claire Pagetti

Etat civil

Née le	06/01/1975
Nationalité	Française
Situation	Vie maritale, deux enfants
e-mail	`claire.pagetti@onera.fr`
web	`http://www.onera.fr/staff/claire-pagetti`

Postes occupés

2013-	**Maître de recherche 1** à l'ONERA
2007-	**MAST** (maître de conférences associé) à l'ENSEEIHT département enseignement TR (Télécommunications et Réseaux) et équipe recherche IRIT - IRT.
2005-2013	**Ingénieur de recherche** à l'ONERA/DTIM centre de Toulouse.
2004-2005	**Post-doctorat** à l'INRIA Futurs, Orsay, dans l'équipe Alchemy (architecture et compilation). Superviseurs : Albert Cohen et Christine Eisenbeis.
2003-2004	**1/2 ATER** au Labri, Université de Bordeaux, dans l'équipe MVT-si (modélisation et vérification des systèmes).
2000-2003	**Monitorat** à l'Ecole Centrale de Nantes.

Diplômes - Formation

2000-2004	**Doctorat en automatique et informatique appliquée** à l'Ecole Centrale de Nantes, soutenu le 20 avril 2004. Directeurs de thèse : Franck Cassez et Olivier Roux. Titre : *Extension temps réel du langage AltaRica*.
1999-2000	**DEA en Informatique** à l'Université de Rennes I, mention assez bien.
1998-1999	**DEA en Mathématiques : Analyse et analyse appliquée** à l'Université de Rennes I.
1997-1998	**Agrégation de Mathématiques**
1994-1997	**Etudes universitaires en Mathématiques** à l'Université de Besançon (Maîtrise de mathématiques mention assez bien, licence de mathématiques mention très bien, DEUG MIAS mention très bien).

Autres activités

2003-2004	Kholles en Math Spé P' - Lycée Montaigne Bordeaux

2001-2002	Présidente de l'association des doctorants A2D-STIM `http://membres.multimania.fr/a2dstim/` (organisation de la première journée de l'école doctorale ED-STIM, organisation de l'assemblée générale de la CJC `http://cjc.jeunes-chercheurs.org/presentation/reunions/2001-10-14/`, membre de la cellule web de la CJC)
1999-2000	Kholles en Math Sup - Lycée Chateaubriand Rennes
1992-2000	Nombreux cours particuliers de mathématiques

7.2 Activités d'administration et responsabilités collectives

Responsable de contrats ONERA

DREAMS : 2013-2017 (projet européen FP7), responsable de la partie ONERA.

WC*T : 2011-2014 (financement Thales)

SHRINK-Reconf : 2008-2012 (financement DPAC), responsable des sous lots SHRINK et Reconf

Scarlett : 2008-2012 (projet européen FP7), responsable de la partie ONERA.

Astac Amil : 2007-2008 (expertise DGA), co-responsable avec Jacques Cazin.

Martiac : 2005-2008 (financement Airbus), co-responsable avec Frédéric Boniol.

Animatrice du séminaire scientifique DTIM

2007-2013 : un séminaire environ toutes les 2 semaines hors vacances scolaires.

Membre de comités de programme

ECRTS : 2015 `http://www.ecrts.org/`

RTAS : 2015 `http://2015.rtas.org/`, WiP RTAS 2013

SAC - track EMBS : 2011 à 2015 `http://retis.sssup.it/sac2015/`

RTNS : 2013, 2014 `http://leat.unice.fr/RTNS2013/#page=home`

MEMOCODE : 2014 `http://memocode.irisa.fr/2014/`

CIIA (track Computer Systems and Applications) : 2013 `http://ciia2013.dzportal.net/`

Symposium MARC : 2012 `http://sites.onera.fr/scc/marconera2012`

Membre de comités d'organisation

Journées FAC : depuis 2008 `http://seminaire-verif.enseeiht.fr/FAC/`

Conférence ERTS : 2012, 2014 `http://www.erts2014.org/`

Workshop MARC 2012 `http://sites.onera.fr/scc/marconera2012`

Ecole jeunes chercheurs ETR : 2013 `http://irit.fr/ETR13/`

Relecture d'articles journaux

2014 : Progress in Aerospace Sciences (jpas) `http://ees.elsevier.com/jpas/`,

2013 : Science of Computer Programming (scp) `http://ees.elsevier.com/scico/`

2011 : Journal of Information Security (EURASIP) `http://jis.eurasipjournals.com/`

Implication GDR ASR

co-responsable avec Pierre Boulet pôle Architecture et systèmes embarqués hautes performances GDR ASR : depuis 2012

Notre rôle consiste à aider et soutenir les actions GDR autour de la thématique des systèmes embarqués.

- Budget annuel : 15 keuros
- 4 actions
- plusieurs écoles de jeunes chercheurs et conférences.

Chantier RTRA Torrents

Responsable du chantier Torrents : depuis 2011

Notre activité consiste à organiser des manifestations scientifiques autour de la thématique de l'implantation sûre de systèmes embarqués temps réel stricts et tolérants aux fautes sur les futures générations de plates-formes (multi/pluri-cœurs sur étagère, architectures distribuées) et à augmenter le rayonnement de la recherche Toulousaine.

— Visibilité : L'ensemble des activités est résumé sur la page web `http://www.irit.fr/torrents`. Les informations sont envoyées sur la mailing list `https://sympa.laas.fr/sympa/info/torrents` (Impact : 30 abonnés directs à la mailing list)
— Séminaires : nous avons un engagement de 4 séminaires annuels. Impact 45 participants.
— Workshop : nous avons un engagement sur l'organisation d'un workshop annuel. Nous invitons 4 personnalités de la recherche et de l'industrie pour présenter les dernières avancées dans une thématique donnée. Impact : 60 participants.
— Rôle : contribution à l'organisation du programme, gestion de la logistique avec Yvonne Le Breton la secrétaire du DTIM (prise en charge des billets, hôtels, soirées de gala et buffets).

7.3 Encadrement

En tant que chercheur à l'ONERA, j'ai assuré (et j'assure) l'encadrement ou co-encadrement scientifique de post-doctorants, d'étudiants en thèse, en Master Recherche et en formation ingénieur.
Tableau récapitulatif :

Type d'encadrement	Nombre
Post-doctorats	3
Thèses soutenues	3 140% taux d'encadrement cumulé
Thèse en cours	3
Master et PFE	7
2ème année ingénieur	2
Participation à jury de thèse	5

Encadrement de post-doctorant (cadre chantier Torrents)

1. Angeliki Kritikakou, Run-time Control to Increase Task Parallelism in Mixed-Critical Systems (2013-2014, avec mon collègue Matthieu Roy LAAS). Financement STAE RTRA.
2. Gilles Lasnier, Simulation de systèmes Cyber Physiques (2012-2013, avec ma collègue Janette Cardoso ISAE). Financement STAE RTRA.
3. Wolfgang Puffitsch, Génération de code multi-périodique sur cible pluri-cœurs (2012-2013). Financement STAE RTRA.

Co-encadrement de travaux de thèses (soutenues)

1. Mikel Cordovilla. Thèse de doctorat de Supaero (spécialité informatique fondamentale et parallélisme) - 2008-2012.
   — Financement ONERA
   — M. Cordovilla est actuellement ingénieur chez Mastercard à Dublin.
   — Taux d'encadrement : 25% Frédéric Boniol, 25% Eric Noulard, 50% Claire Pagetti
   — Environnement de développement d'applications multi-périodiques sur plateforme multi-cœurs.
   Résumé : Les logiciels embarqués critiques de contrôle-commande sont soumis à des contraintes fortes englobant le déterminisme, la correction logique et la correction temporelle. Nous supposons que les spécifications sont exprimées à l'aide du langage formel de description d'architectures logicielles temps réel multi-périodiques Prelude.
   L'objectif de cette thèse est, à partir d'un programme Prelude ou d'un ensemble de tâches temps réel dépendantes, de générer un code multi-tâches exécutable sur une architecture multi-cœurs tout en respectant la sémantique initiale. Pour cela, nous avons développé une boîte à outil, SchedMCore, permettant : 1) d'une part, la vérification formelle de l'ordonnançabilité. La vérification proposée est basée sur le parcours exhaustif du comportement avec pas de temps

discret. Il est alors possible d'analyser des politiques en-ligne (FP, gEDF, gLLF et LLREF) mais également de calculer une affectation de priorité fixe valide et une séquence valide hors-ligne. 2) d'autre part, l'exécution multi-tâches sur une cible multi-cœurs. L'exécutif encode les politiques proposées étudiées dans la partie d'analyse d'ordonnançabilité, à savoir les quatre politiques en-ligne ainsi que les séquences valides générées. 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.
— Rapporteurs : Emmanuel Grolleau (ENSMA) et Olivier H. Roux (IRCCYN Nantes)

2. Michaël Lauer. Thèse de doctorat de l'INP-Toulouse - 2009-2012. Prix de thèse Leopold Escande.
— Financement ANR - projet Satrimmap.
— M. Lauer est actuellement maître de conférences au LAAS.
— Taux d'encadrement : 33% Frédéric Boniol, 33% Jérôme Ermont, 33% Claire Pagetti
— Une méthode globale pour la vérification d'exigences temps réel : application à l'Avionique Modulaire Intégrée.
Résumé : Dans le domaine de l'aéronautique, les systèmes embarqués ont fait leur apparition durant les années 60, lorsque les équipements analogiques ont commencé à être remplacés par leurs équivalents numériques. Dès lors, l'engouement suscité par les progrès de l'informatique fut tel que de plus en plus de fonctionnalités ont été numérisées. L'accroissement permanent de la complexité des systèmes a conduit à la définition d'une architecture appelée Avionique Modulaire Intégrée (IMA pour Integrated Modular Avionics). Cette architecture se distingue des architectures antérieures, car elle est fondée sur des standards (ARINC 653 et ARINC 664 partie 7) permettant le partage des ressources de calcul et de communication entre les différentes fonctions avioniques. Ce type d'architecture est appliqué aussi bien dans le domaine civil avec le Boeing B777 et l'Airbus A380, que dans le domaine militaire avec le Rafale ou encore l'A400M. Pour des raisons de sûreté, le comportement temporel d'un système s'appuyant sur une architecture IMA doit être prévisible. Ce besoin se traduit par un ensemble d'exigences temps réel que doit satisfaire le système. Le problème exploré dans cette thèse concerne la vérification d'exigences temps réel dans les systèmes IMA. Ces exigences s'articulent autour de chaînes fonctionnelles, qui sont des séquences de fonctions. Une exigence spécifie alors une borne acceptable (minimale ou maximale) pour une propriété temporelle d'une ou plusieurs chaînes fonctionnelles. Nous avons identifié trois catégories d'exigences temps réel, que nous considérons pertinentes vis-à-vis des systèmes étudiés. Il s'agit des exigences de latence, de fraîcheur et de cohérence. Nous proposons une modélisation des systèmes IMA, et des exigences qu'ils doivent satisfaire, dans le formalisme du tagged signal model. Nous montrons alors comment, à partir de ce modèle, nous pouvons générer pour chaque exigence un programme linéaire mixte, c'est-à-dire contenant à la fois des variables entières et réelles, dont la solution optimale permet de vérifier la satisfaction de l'exigence.
— Rapporteurs : Frédéric Mallet (INRIA Sophia Antipolis) et Pascal Minet (INRIA Rocquencourt)

3. Julien Forget. Thèse de doctorat de Supaero (spécialité informatique fondamentale et parallélisme) - 2006-2009.
— Financement Astrium.
— J. Forget est aujourd'hui maître de conférences au LIFL.
— Taux d'encadrement : 40% Frédéric Boniol, 60% Claire Pagetti
— Programmation et implantation de systèmes contrôle-commande distribués.
Résumé : L'objectif de cette thèse était de définir un langage de programmation formel, appelé Prelude, répondant aux exigences des systèmes embarqués temps réel de type « contrôle-commande ». Le langage Prelude permet de décrire un assemblage multi-périodique de fonctions importées, supposées écrites dans un autre langage (Scade, C ou Ada). La base sémantique est celle du synchrone mais les horloges sont rythmées par des entiers représentant le temps réel et non plus des horloges booléennes. De plus, les flots sont calculés selon l'hypothèse *synchrone relâchée* c'est-à-dire avant la prochaine échéance. Une spécification Prelude peut ensuite être compilée en un système multi-tâches gérées par un ordonnanceur temps réel. Une chaîne de compilation complète allant de programmes Prelude et de fonctions importées en C vers le système d'exploitation mono-processeur Marte OS a été développée en Ocaml.
— Rapporteurs : Alain Girault (INRIA Grenoble) et Yvon Trinquet (IRCCyN Nantes).

Co-encadrement de travaux de thèses (en cours)

1. Rémy Wyss. 2010 -
   — Financement ONERA
   — Arrêt maladie depuis août 2012.
   — Taux d'encadrement 50% Frédéric Boniol, 50% Claire Pagetti
   — Atelier d'aide à la conception de systèmes multi-périodiques.
   Résumé : L'objectif de la thèse est de fournir un ensemble d'outils de validation de spécification en Prelude, ainsi qu'une méthodologie de développement associée au langage. Concernant la validation à partir d'une spécification Prelude, il est possible de vérifier formellement un certain nombre de propriétés temporelles quantitatives avant la phase d'implantation. Les propriétés intéressantes pour des applications de type contrôle-commande sont : la latence entre une entrée et son effet produit en sortie du système, la fraîcheur de données consommées et la cohérence entre données. Le deuxième objectif de la thèse est de permettre des spécifications partielles en utilisant un nouvel opérateur « don't care » et de laisser le compilateur généré des programmes complets satisfaisant les propriétés mentionnées précédemment.
2. Quentin Perret. mars 2014 -
   — Financement CIFRE Airbus
   — Taux d'encadrement 50% Eric Noulard, 25% Claire Pagetti, 25% Pascal Sainrat
   — Langage et méthode pour code prédictible sur pluri-cœurs
   Résumé : L'objectif de la thèse est d'étudier puis de proposer un modèle de programmation et une méthodologie adaptés à l'exécution déterministe sur processeurs pluri-cœurs visant l'embarqué critique. L'idée est de considérer qu'une application embarquée critique du futur sera naturellement plongée dans un système multi-processeurs communicant de façon explicite (i.e. avec peu de partage implicite) mais probablement de manière beaucoup plus efficace que sur un réseau embarqué classique actuel type AFDX ou CAN. La cible choisie est le MPPA de Kalray.
3. Kevin Delmas. novembre 2014 -
   — Financement ONERA
   — Taux d'encadrement 60% Rémi Delmas, 40% Claire Pagetti
   — Méthode et environnement de développement générique pour applications prédictibles et tolérantes aux fautes sur processeurs pluri-cœurs
   Résumé : Les prochaines générations de processeurs seront des pluri-cœurs, c'est-à-dire des puces constituées de plusieurs processeurs reliés par un réseau sur puce. Nous nous intéresserons dans le cadre de cette thèse à deux catégories de propriétés : les propriétés temps réel d'une part, et les propriétés de tolérance d'une application aux fautes. Du point de vue temps réel, le concepteur doit démontrer le respect d'exigences temporelles. Du point de vue tolérance aux fautes, le concepteur doit démontrer, entre autres, que l'exécution de l'application est robuste aux fautes issues du matériel et en particulier celles dues aux SEU (Single Event Upset). Dans le cas d'un SEU, des bits stockant l'information d'un programme où ses instructions sont potentiellement altérés par collision d'une particule de haute énergie avec le processeur ou la mémoire d'un calculateur.
   L'objectif de la thèse sera de développer des méthodes et outils génériques permettant la programmation temps réel sur ce type d'architecture, et exploitant la multiplicité des unités de calcul disponibles pour assurer la redondance des calculs nécessaires pour tolérer les fautes de type SEU.

Encadrement de travaux de master recherche / PFE

1. Kevin Delmas, 5ème année école d'ingénieur INSA Toulouse, « Contrôleur longitudinal tolérant aux fautes », 2013-2014, avec mon collègue Rémi Delmas ;
2. Romain Gratia, 5ème année école d'ingénieur ENSEM (Ecole Nationale Supérieure d'Electricité et de Mécanique) à Nancy, « Programmation d'applications temps réel sur pluri-cœurs », 2012-2013, avec mon collègue Eric Noulard ;
3. Guillaume Brau, Master recherche informatique UPS (Université Paul Sabatier Toulouse), « Comparaison d'architectures embarquées », 2011-2012 ;
4. Kushal Gupta, Master AESS ISAE, « Langage de description de reconfiguration », 2010-2011, avec mon collègue Eric Noulard ;

5. Adrien Charles, 5ème année école d'ingénieur Ecole des Mînes d'Albi, « A C library for multi-threaded programs on a multi-core architecture », 2009-2010, avec mon collègue Eric Noulard ;
6. Mikel Cordovilla, Master CAMSI UPS/ENSEEIHT Toulouse, « Génération d'ordonnancements préemptifs critiques », 2007-2008, avec mon collègue Frédéric Boniol ;
7. Xavier Dumas, Master ISC Bordeaux, « Model Transformation : from AADL to AltaRica », 2006-2007, avec mes collègues Philippe Dhaussy (ENSIETA) et Laurent Sagaspe.

Encadrement de stage ingénieur 2ème année

1. Julie Baro, 2ème année ENSMA/ISAE, 2010-2011
2. Ting Liu, 2ème année ENSEEIHT, filière informatique, 2010-2011

Participation à des jurys de thèse

(Remarque : hors étudiants que j'ai co-encadrés).

1. Membre examinateur du jury de thèse de doctorat de l'Université de Paris Sud de Mme Jinyi Bin (juillet 2014).
2. Membre examinateur du jury de thèse de doctorat du CEA de M Ernest Wozniak (juillet 2014).
3. Membre examinateur du jury de thèse de doctorat de l'Ecole Centrale de Nantes de Melle Nadine Abdallah (février 2014).
4. Membre examinateur du jury de thèse de doctorat de l'Ecole Nationale Supérieure de Télécommunication de Brest (ENSTB) de M. Jean Charles Roger (décembre 2006).
5. Membre examinateur du jury de thèse de doctorat de Supaero M. Wolfgang Theurer (décembre 2006).

7.4 Enseignement

J'ai enseigné dans le cadre d'un monitorat à l'Ecole Centrale de Nantes (64 h équivalent TD pendant 3 ans), suivi d'une année de demi-ATER à l'Université de Bordeaux (96h). A mon arrivée à l'ONERA (2005-2007), j'ai effectué des vacations à Supaéro, l'INSA et l'ENSEEIHT. Depuis 2007, j'effectue mon enseignement principalement à l'ENSEEIHT en tant que maître de conférences associé à mi temps (MAST). Je donne donc en moyenne 96h équivalent TD heures annuelles.

7.5 Productions scientifiques

Tableau récapitulatif :

Type de production	nombre
Journaux	5
Conférences A+	1
Conférences A	6
Conférences B	8
Conférences autre	7

7.5.1 Liste des publications

Les bases de données représentatives dans mon domaine de recherche sont :
— GoogleScholar `http://scholar.google.fr/citations?user=xiDJUZIAAAAJ&hl=en&oi=ao`
— la base de données Microsoft `http://www.journalogy.net/Author/755664/claire-pagetti`
— la base DBPL `http://www.informatik.uni-trier.de/~ley/pers/hd/p/Pagetti:Claire.html`
Utilisation du CORE Ranking of Conferences and Journals in Computer Science `http://clip.dia.fi.upm.es/~clip/VenueImpact/index.html` pour déterminer les rangs des conférences et des journaux.

Brevet

[JTA12] Victor Jegu, Benoît Triquet, Frédéric Aspro, Claire Pagetti et Frédéric Boniol. *Procédé et dispositif de chargement et d'exécution d'instructions à cycles déterministes dans un système avionique multi-coeurs ayant un bus dont le temps d'accès est non prédictible.* EP Patent App. EP20,100,734,208. Date de priorité 5 juin 2009. 2012.

Revues internationales

[LBPE14] Michaël Lauer, Frédéric Boniol, Claire Pagetti et Jérôme Ermont. "End-to-end latency and temporal consistency analysis in networked real-time systems". In : *International Journal of Critical Computer-Based Systems (IJCCBS)* 5.3/4 (2014), p. 172–196.

[BBB12] Pierre Bieber, Frédéric Boniol, Marc Boyer, Eric Noulard et Claire Pagetti. "New Challenges for Future Avionic Architectures". In : *Journal AerospaceLab* 4 (2012).

[PFB11] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla et David Lesens. "Multi-task Implementation of Multi-periodic Synchronous Programs". In : *Discrete Event Dynamic Systems* 21.3 (2011), p. 307–338.

[BEP09] Frédéric Boniol, Jérôme Ermont et Claire Pagetti. "Verification of real-time systems with preemption : negative and positive results". In : *Innovations in Systems and Software Engineering (ISSE)* 5.3 (2009), p. 163–179.

[CPR04] Franck Cassez, Claire Pagetti et Olivier Roux. "A Timed Extension for ALTARICA". In : *Fundamenta Informaticae* 62.3-4 (2004), p. 291–332.

Revues nationales

[TBDP07] Wolfgang Theurer, Frédéric Boniol, Philippe Dhaussy et Claire Pagetti. "Un cadre conceptuel pour la modélisation multi point de vue de systèmes embarqués". In : *L'OBJET* 13.2-3 (2007), p. 79–110.

Conférences internationales avec comité de relecture et actes

2014.

[DDP14] Kevin Delmas, Rémi Delmas et Claire Pagetti. "Automatic architecture hardening using safety patterns". In : *4th International Symposium on Model-Based Safety and Assessment (MBSA'14)*. 2014.

[DFG14] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti et Wolfgang Puffitsch. "Predictable Flight Management System Implementation on a Multicore Processor". In : *Proceedings of the 7th Conference on Embedded Real Time Software and Systems (ERTS'14)*. 2014.

[KBP14] Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange et Matthieu Roy. "Run-time Control to Increase Task Parallelism in Mixed-Critical Systems". In : *26th Euromicro Conference on Real-Time Systems (ECRTS'14)*. Juil. 2014, p. 119–128.

[KPR14] Angeliki Kritikakou, Claire Pagetti, Christine Rochange, Matthieu Roy, Madeleine Faugère, Sylvain Girbal et Daniel Gracia Pérez. "Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems". In : *Proceedings of the 22th International Conference on Real-Time and Network Systems (RTNS'14)*. 2014, p. 139–148.

[PSG14] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard et Pierre Siron. "The ROSACE Case Study : From Simulink Specification to Multi/Many-Core Execution". In : *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*. Avr. 2014.

[SPB14] Luca Santinelli, Wolfgang Puffitsch, Frédéric Boniol, Claire Pagetti, Arnaud Dumérat et Victor Jégu. "Grouping Approach to Task Scheduling with Functional and Non-Functional Requirements". In : *Proceedings of the 4th Conference on Embedded Real Time Software and Systems (ERTS'14)*. 2014.

2013.

[BLPE13] Frédéric Boniol, Michaël Lauer, Claire Pagetti et Jérôme Ermont. “Freshness and Reactivity Analysis in Globally Asynchronous Locally Time-Triggered Systems”. In : *Proceedings of the 5th International Symposium NASA Formal Methods (NFM'13)*. 2013, p. 93–107.

[LCS13] Gilles Lasnier, Janette Cardoso, Pierre Siron, Claire Pagetti et Patricia Derler. “Distributed Simulation of Heterogeneous and Real-Time Systems”. In : *Proceedings of the 17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT'13)*. 2013, p. 55–62.

[PNP13] Wolfgang Puffitsch, Eric Noulard et Claire Pagetti. “Mapping a multi-rate synchronous language to a many-core processor”. In : *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*. 2013, p. 293–302.

[WBPF13] Rémy Wyss, Frédéric Boniol, Claire Pagetti et Julien Forget. “End-to-end latency computation in a multi-periodic design”. In : *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*. 2013, p. 1682–1687.

2012.

[BBC12] Julie Baro, Frédéric Boniol, Mikel Cordovilla, Eric Noulard et Claire Pagetti. “Off-line (Optimal) multiprocessor scheduling of dependent periodic tasks”. In : *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*. 2012, p. 1815–1820.

[BCNP12] Frédéric Boniol, Hugues Cassé, Eric Noulard et Claire Pagetti. “Deterministic Execution Model on COTS Hardware”. In : *Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS'12)*. 2012, p. 98–110.

[PBB12] Claire Pagetti, Pierre Bieber, Julien Brunel, Kushal Gupta, Eric Noulard, Thierry Planche, Francois Vialard, Clément Ketchedji, Bernard Bésinet et Philippe Despres. “Reconfigurable IMA platform : from safety assessment to test scenarios on the Scarlett demonstrator”. In : *Proceedings of the 6th Conference on Embedded Real Time Software and Systems and Software (ERTS'12)*. 2012.

[WBFP12a] Rémy Wyss, Frédéric Boniol, Julien Forget et Claire Pagetti. “A Synchronous Language with Partial Delay Specification for Real-Time Systems Programming”. In : *Proceedings of the 10th Asian Symposium on Programming Languages and Systems (APLAS'12)*. 2012, p. 223–238.

2011.

[CBF11] Mikel Cordovilla, Frédéric Boniol, Julien Forget, Eric Noulard et 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)*. 2011, p. 107–116.

[CBNP11] Mikel Cordovilla, Frédéric Boniol, Eric Noulard et Claire Pagetti. “Multiprocessor schedulability analyser”. In : *Proceedings of the 26th Annual ACM Symposium on Applied Computing (SAC'11)*. 2011, p. 735–741.

[FGPR11] Julien Forget, Emmanuel Grolleau, Claire Pagetti et Pascal Richard. “Dynamic priority scheduling of periodic tasks with extended precedences”. In : *Proceedings of the 16th IEEE Conference on Emerging Technologies & Factory Automation (ETFA'11)*. 2011, p. 1–8.

[LEBP11a] Michaël Lauer, Jérôme Ermont, Frédéric Boniol et Claire Pagetti. “Latency and freshness analysis on IMA systems”. In : *Proceedings of the 16th IEEE Conference on Emerging Technologies & Factory Automation (ETFA'11)*. 2011, p. 1–8.

[LEBP11b] Michaël Lauer, Jérôme Ermont, Frédéric Boniol et Claire Pagetti. “Worst Case Temporal Consistency in Integrated Modular Avionics Systems”. In : *13th IEEE International Symposium on High-Assurance Systems Engineering (HASE'11)*. 2011, p. 212–219.

2010.

[FBG10] Julien Forget, Frédéric Boniol, Emmanuel Grolleau, David Lesens et Claire Pagetti. "Scheduling Dependent Periodic Tasks without Synchronization Mechanisms". In : *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'10)*. 2010, p. 301–310.

[FBLP10] Julien Forget, Frédéric Boniol, David Lesens et Claire Pagetti. "A real-time architecture design language for multi-rate embedded control systems". In : *Proceedings of the 25th ACM Symposium on Applied Computing (SAC'10)*. ACM, 2010, p. 527–534.

[LEPB10] Michaël Lauer, Jérôme Ermont, Claire Pagetti et Frédéric Boniol. "Analyzing End-to-End Functional Delays on an IMA Platform". In : *Proceedings of the 4th International Symposium on Leveraging Applications (ISOLA'10)*. 2010, p. 243–257.

2008.

[BHP08] Frédéric Boniol, Pierre-Emmanuel Hladik, Claire Pagetti, Frédéric Aspro et Victor Jégu. "A Framework for Distributing Real-Time Functions". In : *Proceedings of the 6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'08)*. 2008, p. 155–169.

[FBLP08] Julien Forget, Frédéric Boniol, David Lesens et Claire Pagetti. "A multi-periodic synchronous data-flow language". In : *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE'08)*. 2008.

[FBL08] Julien Forget, Frédéric Boniol, David Lesens, Claire Pagetti et Marc Pouzet. "Programming Languages For Hard Real-Time Embedded Systems". In : *Proceedings of the 4th Conference on Embedded Real Time Software and Systems and Software (ERTS'08)*. 2008.

2007.

[BPR07] Frédéric Boniol, Claire Pagetti et François Revest. "Formal Functionally Deterministic Scheduling". In : *Workshop On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA'07 - SHORT PAPER)*. 2007, p. 33–40.

2006.

[BP06] Manuel Baclet et Claire Pagetti. "Around Hopcroft's Algorithm". In : *Proceedings of the 11th International Conference on Implementation and Application of Automata (CIAA'06)*. 2006, p. 114–125.

[CDE06] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau et Marc Pouzet. "*N*-synchronous Kahn networks : a relaxed model of synchrony for real-time systems". In : *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'06)*. 2006, p. 180–193.

2005.

[CDE05] Albert Cohen, Marc Duranton, Christine Eisenbeis, Claire Pagetti, Florence Plateau et Marc Pouzet. "Synchronization of periodic clocks". In : *Proceedings of th 5th ACM International Conference On Embedded Software (EMSOFT'05 - SHORT PAPER)*. 2005, p. 339–342.

2004.

[AP04] Michaël Adélaïde et Claire Pagetti. "On the Urgency Expressiveness". In : *24th International Conference on Foundations of Software Technology and Theoretical Computer Science (FTTCS'04)*. 2004, p. 71–83.

Conférences nationales avec comité de relecture et actes

[LCPS13] Gilles Lasnier, Janette Cardoso, Claire Pagetti et Pierre Siron. *Environnement de coopération de simulation pour la conception de systèmes cyber-physiques*. Actes du colloque francophone MSR 2013 publié dans Journal européen des systèmes automatisés (JESA). Mai 2013.

[WBFP13] Rémy Wyss, Frédéric Boniol, Julien Forget et Claire Pagetti. *Propriétés de latence, fraîcheur et réactivité dans un programme synchrone multi-périodique*. Journées sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2013). Mar. 2013.

[WBFP12b] Rémy Wyss, Frédéric Boniol, Julien Forget et Claire Pagetti. *Calcul de latences dans un programme Prelude*. Proceedings des Journées sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2012). Jan. 2012.

[LBEP11] Michaël Lauer, Frédéric Boniol, Jérôme Ermont et Claire Pagetti. *Analyse de latence et fraîcheur pire cas sur systèmes avioniques modulaires intégrés*. Actes du colloque francophone MSR 2011 publié dans Journal européen des systèmes automatisés (JESA) 45 :1-3. 2011.

[BCF09] Frédéric Boniol, Mikel Cordovilla, Julien Forget, David Lesens et Claire Pagetti. *Implantation multitâche de programmes synchrones multipériodiques*. Actes du colloque francophone MSR 2009 publié dans Journal européen des systèmes automatisés (JESA) 43. 2009.

[DPS08] Xavier Dumas, Claire Pagetti, Laurent Sagaspe, Pierre Bieber et Philippe Dhaussy. *Vers la génération de modèles de sûreté de fonctionnement*. 2ème Conférence Francophone sur les Architectures Logicielles (CAL 2008). 2008.

Autre matériel publié

[KBP13] Angeliki Kritikakou, Olivier Baldellon, Claire Pagetti, Christine Rochange, Matthieu Roy et Fabian Vargas. “Monitoring on-line timing information to support mixed-critical workloads”. In : *Proceedings of the Work-in-Progress Session of the 34th IEEE Real-Time Systems Symposium (WiP-RTSS 2013)*. 2013.

[LB13] Claire Pagetti Luca Santinelli Wolfgang Puffitsch et Frederic Boniol. “Scheduling with Functional and Non-Functional Requirements : the Sub-Functional Approach”. In : *Proceedings of the Work-in-Progress Session of the 25th Euromicro Conference on Real-Time Systems (WiP-ECRTS 2013)*. 2013.

[BP12] Guillaume Brau et Claire Pagetti. “TTEthernet-based architecture simulation with Ptolemy II”. In : *Proceedings of the 6th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2012)*. Pont-à-Mousson, France, nov. 2012, p29–32. URL : http://hal.archives-ouvertes.fr/hal-00909115.

[BPLE12] Frédéric Boniol, Claire Pagetti, Michael Lauer et Jérôme Ermont. “End-to-end latency analysis in networked real-time systems”. In : *Proceedings of the 6th International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2012)*. Electronic Workshops in Computing (eWiC). BCS, sept. 2012.

[dBNP11] Bruno d'Ausbourg, Marc Boyer, Eric Noulard et Claire Pagetti. “Deterministic Execution on Many-Core Platforms : application to the SCC”. In : *Proceedings of the 4th Many-core Applications Research Community Symposium (MARC 2011)*. Potsdam University, 2011, p. 43–48. ISBN : 978-3-86956-169-1.

[BNB10] Pierre Bieber, Eric Noulard, Julien Brunel, Claire Pagetti, Thierry Planche et François Vialard. “Preliminary design of future reconfigurable IMA platforms - Safety assessment”. In : *Proceedings of the 27th Congress International Council of the Aeronautical Sciences (ICAS 2010)*. 2010.

[BNP09] Pierre Bieber, Eric Noulard, Claire Pagetti, Thierry Planche et Francois Vialard. “Preliminary design of future reconfigurable IMA platforms”. In : *Proceedings of the workshop APRES - SIGBED Review*. 2009, p. 7.

[FBLP09] Julien Forget, Frédéric Boniol, David Lesens et Claire Pagetti. “Implementing Multi-Periodic Critical Systems : from Design to Code Generation”. In : *Proceedings FM-09 Workshop on Formal Methods for Aerospace*. 2009, p. 34–48.

7.5.2 Liste des présentations invitées

2013

1. Claire Pagetti, Pierre Bieber, and Virginie Wiels. Certification and Safety Analysis Challenges in the Avionic Domain. Tutorial sur « Mixed-Criticality Systems : Design and Certification Challenges » à l'Embedded Systems Week `http://www.tik.ee.ethz.ch/~nikolays/mixed_criticality_tutorial.html`
2. Keynote sur « Developing multi-periodic critical embedded systems on multi/many-core architectures » au 6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems (CRTS 2013) `http://www.mrtc.mdh.se/CRTS2013/index.php?choice=invited`
3. Exposé sur « Ordonnancement multiprocesseur » au groupe de travail CCT CNES SIL (Architecture des Systèmes Informatiques et Génie Logiciel).
4. Exposé sur « Architecture multi/many-coeur pour les systèmes critiques" (projet TOAST) » à la quatrième édition du séminaire "Ingénierie des Systèmes Complexes à Logiciel Prépondérant" organisé par DGA Techniques Aéronautiques `http://www.ixarm.com/Seminaire-ISCLP-Methodologies-et`

7.5.3 Liste des rapports contractuels

2014

1. Guy Durrieu, Claire Pagetti. Rapports D2.2 et D2.3 : Maîtrise des pires-temps sur architecture multi-coeurs. Contrat Thales WC*T.
2. Virginie Wiels,Alexandre Amiez, Rémi Delmas, Piere-Loïc Garoche, Thomas Loquen, Eric Noulard, Claire Pagetti. Rapport sur la conception sûre de systèmes de contrôle commande. Projet ONERA Forces 3.

2013

1. Guy Durrieu, Eric Noulard, Claire Pagetti, Wolfgang Puffitsch. Rapports D1 et D2.1 : Maîtrise des pires-temps sur architecture multi-coeurs. Contrat Thales WC*T.

2011

1. Pierre Bieber, Claire Pagetti. Contribution à plusieurs rapports sur la certification Scarlett.
2. Pierre Bieber, Frédéric Boniol, Julien Brunel, Claire Pagetti. Rapports d'avancement DPAC : DPAC reconfiguration et DPAC Shrink.

2010

1. Frédéric Boniol, Claire Pagetti. Proposition de langage intermédiaire pour l'architecture SHRINK. Contrat DPAC/SHRINK.
2. Pierre Bieber, Julien Brunel, Claire Pagetti. Safety Assessment of the Reconfguration System. Projet européen Scarlett.

2009

1. Julien Forget, Frédéric Boniol, David Lesens, Claire Pagetti, Programming and implementing Control-Command systems, progression report 4. Contrat Astrium thèse Julien Forget.
2. Frédéric Boniol, Michel Lemaître, Claire Pagetti, Gérard Verfaillie. Outil de placement pour l'architecture SHRINK. Contrat DPAC/SHRINK.

2008

1. Julien Forget, Frédéric Boniol, David Lesens, Claire Pagetti, Programmation et implantation de systèmes de Contrôle-Commande distribués. Rapport d'avancement 3 Contrat Astrium thèse Julien Forget.
2. Frédéric Boniol, Claire Pagetti, Jérôme Ermont. Etude de la structuration des messages et son impact sur les performances de l'architecture MARTIAC. Contrat Airbus Martiac.

3. Frédéric Boniol, Marc Boyer, David Doose, Claire Pagetti. Rapports ADCN Tâches 5.2.2 et 5.2.3. Contrat Airbus ADCN+.
4. Claire Pagetti et les partenaires Satrimmap. Satrimmap - WP1 - Tâche 1.1. - Besoins des fonctions avioniques. 2008. Projet ANR Satrimmap.
5. Jacques Cazin, Claire Pagetti. Astac-Amil - Application des solutions technologiques et d'architectures du monde civil aux avioniques militaires intégrées. Rapport final - Poste 3. Contrat d'expertise DGA Astac-Amil.

2007

1. Henri Bauer, Frédéric Boniol, Marc Boyer, Jean-Loup Bussenot, David Doose, Christian Fraboul, Jérôme Ermont, Claire Pagetti, Jean-Luc Scharbarg, Boris Sidoruk. Plusieurs rapports ADCN+. Contrat Airbus ADCN+.
2. Claire Pagetti, Frédéric Boniol, Marc Boyer, Jérôme Ermont, Jean-Luc Scharbarg, Christine Rochange, Pascal Sainrat. Plusieurs rapports MARTiAC. Contrat Airbus Martiac.
3. Jacques Cazin, Claire Pagetti, Yvon Trinquet, Sébastien Faucou, Yannick Chevalier, Nicolas Riviere, Agnan de Bonneval, Ahlem Mifdaoui, Fabrise Frances. Astac-Amil : Application des solutions technologiques et d'architectures du monde civil aux avioniques militaires intégrées. Poste 2 : Etat de l'art recensement et caractérisation. Contrat d'expertise DGA Astac-Amil.
4. Pierre Bieber, Patrice Cros, Guy Durrieu, Pierre Michel, Claire Pagetti, Christel Seguin, Boris Sidoruk, Virginie Wiels. Systèmes embarqués. Projet ONERA SE.

2006

1. Muriel Brunet, Patrick Le Blaye, Claire Pagetti, Christel Seguin. Qualitative Safety Assessment. Projet européen FP7 Flysafe.
2. Claire Pagetti, Frédéric Boniol. Plusieurs rapports Martiac. Contrat Airbus Martiac.
3. Christel Seguin, Claire Pagetti, OFFIS, University of York. Aircraft Supportability : A survey of trends, challenges and opportunities. Contrat Depnet.

2005

1. Robert Demolombe, Gérard Eizenberg, Jack Foisseau, Claire Pagetti, Claire Saurel, Guy Zanon. Techniques de base pour l'ingénierie des systèmes de systèmes. Etat de l'art, concepts de base et programme de travail. Projet ONERA MISS

Bibliographie

[AS99] Tarek F. Abdelzaher et Kang G. Shin. "Combined Task and Message Scheduling in Distributed Real-Time Systems". In : *IEEE Trans. Parallel Distrib. Syst.* 10.11 (1999), p. 1179–1191.

[ABD13] Andreas Abel, Florian Benz, Johannes Doerfert, Barbara Dörr, Sebastian Hahn, Florian Haupenthal, Michael Jacobs, Amir H. Moin, Jan Reineke, Bernhard Schommer et Reinhard Wilhelm. "Impact of Resource Sharing on Performance and Performance Prediction : A Survey". In : *24th International on Conference Concurrency Theory (CONCUR 2013)*. 2013.

[Abs] Absint. *aiT Worst-Case Execution Time Aanalyzers*. url : `http://www.absint.com/ait/`.

[Aer05] Aeronautical Radio Inc. *Aircraft Data Network Part 7 : "Avionics Full Duplex Switched Ethernet (AFDX) Network"*. 2005.

[Aer97] Aeronautical Radio Inc. *Avionics Application Software Standard Interface*. 1997.

[ACGR09] Mouaiad Alras, Paul Caspi, Alain Girault et Pascal Raymond. "Model-Based Design of Embedded Control Systems by Means of a Synchronous Intermediate Model". In : *International Conference on Embedded Software and Systems (ICESS'09)*. Hangzhou, China, mai 2009.

[AFM02] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson et 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. London, UK, UK : Springer-Verlag, 2002, p. 460–464. isbn : 3-540-43419-4. url : `http://dl.acm.org/citation.cfm?id=646486.694613`.

[ABB09] James H. Anderson, Sanjoy K. Baruah et Björn B. Brandenburg. "Multicore Operating-System Support for Mixed Criticality". In : *Proceedings of the Workshop on Mixed Criticality : Roadmap to Evolving UAV Certification (WMC'09)*. Avr. 2009.

[And05] Charles André. *Comparaison des styles de programmation de langages synchrones*. Rapp. tech. RR-2005-13. Sophia-Antipolis, (F) : I3S, Juin 2005.

[Aud91] Neil C. Audsley. *Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times*. Rapp. tech. YCS 164. Dept. Computer Science, University of York, déc. 1991.

[BB07] Theodore P. Baker et Sanjoy K. Baruah. "Schedulability analysis of multiprocessor sporadic task systems". In : *Handbook of Realtime and Embedded Systems*. CRC Press, 2007.

[BCRS10] Clément Ballabriga, Hugues Cassé, Christine Rochange et Pascal Sainrat. "OTAWA : An Open Toolbox for Adaptive WCET Analysis". In : *8th International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS'10)*. T. 6399. LNCS. 2010, p. 35–46.

[Bar12] Sanjoy Baruah. "Semantics-preserving implementation of multirate mixed-criticality synchronous programs". In : *20th International Conference on Real-Time and Network Systems (RTNS'12)*. Pont á Mousson, France, 2012, p. 11–19. isbn : 978-1-4503-1409-1. doi : `10.1145/2392987.2392989`.

[BCLS14] Sanjoy K. Baruah, Bipasa Chattopadhyay, Haohan Li et Insik Shin. "Mixed-criticality scheduling on multiprocessors". In : *Real-Time Systems* 50.1 (2014), p. 142–177.

[BF11] Sanjoy K. Baruah et Gerhard Fohler. "Certification-Cognizant Time-Triggered Scheduling of Mixed-Criticality Systems". In : *Proceedings of the 32nd IEEE Real-Time Systems Symposium, (RTSS'11)*. 2011, p. 3–12.

[BV08] Sanjoy K. Baruah et Steve Vestal. "Schedulability Analysis of Sporadic Tasks with Multiple Criticality Specifications". In : *Proceedings of the 20th Euromicro Conference on Real-Time Systems (ECRTS'08)*. 2008, p. 147–155.

[BLS10] Sanjoy Baruah, Haohan Li et Leen Stougie. "Towards the Design of Certifiable Mixed-criticality Systems". In : *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '10)*. 2010, p. 13–22.

[AZDG13] Zaid Al-bayati, Haibo Zeng, Marco Di Natale et Zonghua Gu. "Multitask implementation of synchronous reactive models with Earliest Deadline First scheduling". In : *8th IEEE International Symposium on Industrial Embedded Systems (SIES 2013)*. 2013, p. 168–177.

[BDL06] Gerd Behrmann, Alexandre David, Kim G. Larsen, John Håkansson, Paul Pettersson, Wang Yi et Martijn Hendriks. "UPPAAL 4.0". In : *Proceedings of the 3rd International Conference on the Quantitative Evaluation of SysTems (QEST) 2006*. IEEE Computer Society. 2006, p. 125–126. isbn : 0-7695-2665-9.

[BLR05] Gerd Behrmann, Kim G. Larsen et Jacob I. Rasmussen. "Optimal scheduling using priced timed automata". In : *SIGMETRICS Perform. Eval. Rev.* 32 (4 mar. 2005), p. 34–40. issn : 0163-5999.

[BCE03] Albert Benveniste, Paul Caspi, Stephen Edwards, Nicolas Halbwachs, Paul Leguernic et Robert de Simone. *Synchronous Languages, 12 Years Later*. Proceedings of the IEEE. Jan. 2003.

[BBF08] Bernard Berthomieu, Jean-Paul Bodeveix, Patrick Farail, Mamoun Filali, Hubert Garavel, Pierre Gaufillet, Frederic Lang et François Vernadat. "Fiacre : an Intermediate Language for Model Verification in the Topcased Environment". In : *Proceedings of the 4th Conference on Embedded Real Time Software and Systems and Software (ERTS2'08)*. Toulouse, France, 2008.

[BF14] Antoine Bertout et Julien Forget. "A heuristic to minimize the cardinality of a real-time task set by automated task clustering". In : *Proceedings of the 29th ACM Symposium on Applied Computing (SAC'14)*. 2014, p. 1431–1436.

[BFO14] Antoine Bertout, Julien Forget et Richard Olejnik. "Minimizing a real-time task set through Task Clustering". In : *22nd International Conference on Real-Time Networks and Systems, (RTNS'14)*. 2014, p. 23.

[BBP13] Emiliano Betti, Stanley Bak, Rodolfo Pellizzoni, Marco Caccamo et Lui Sha. "Real-Time I/O Management System with COTS Peripherals". In : *IEEE Trans. Computers* 62.1 (2013), p. 45–58.

[Bin14] Jingyi Bin. "Controlling Execution Time Variability using COTS for Safety-critical Systems". PhD. Université Paris-sud XI, 2014.

[BGG14] Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, Arnaud Grasset et Alain Merigot. "Studying co-running avionic real-time applications on multi-core COTS architectures". In : *Embedded Real Time Software and Systems (ERTS'14)*. 2014.

[BB04] Enrico Bini et Giorgio C. Buttazzo. "Biasing Effects in Schedulability Measures". In : *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS'04)*. 2004, p. 196–203.

[BJF12] Simon Bliudze, Mathieu Jan et Xavier Fornari. "From Model-Based to Real-Time Execution of Safety-Critical Applications : Coupling Scade with OASIS". In : *Embedded Real Time Software and Systems (ERTS'012)*. 2012.

[BBB10] Armelle Bonenfant, Ian Broster, Clément Ballabriga, Guillem Bernat, Hugues Cassé, Michael Houston, Nicholas Merriam, Marianne de Michiel, Christine Rochange et Pascal Sainrat. *Coding guidelines for WCET analysis using measurement-based and static analysis techniques*. anglais. Rapport de recherche IRIT/RR–2010-8–FR. Université Paul Sabatier, Toulouse : IRIT, mar. 2010. url : http://www.irit.fr/publis/TRACES/IRIT-RR--2010-8--FR.pdf.

[Bor05] Shekhar BORKAR. "Designing Reliable Systems from Unreliable Components : The Challenges of Transistor Variability and Degradation". In : *IEEE Micro* 25.6 (2005), p. 10–16.

[BT13] Adnan BOUAKAZ et Jean-Pierre TALPIN. "Buffer minimization in earliest-deadline first scheduling of dataflow graphs". In : *SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES'13)*. 2013, p. 133–142.

[BA07] Björn B. BRANDENBURG et James H. ANDERSON. "Integrating Hard/Soft Real-Time Tasks and Best-Effort Jobs on Multiprocessors". In : *Proceedings of thr 19th Euromicro Conference on Real-Time Systems, (ECRTS'07)*. 2007, p. 61–70.

[BB13] Alan BURNS et Sanjoy BARUAH. "Towards A More Practical Model for Mixed Criticality Systems". In : *Proceedings of the Workshop on Mixed Criticality (WMC'13)*. 2013, p. 1–6.

[BB11] Alan BURNS et Sanjoy K. BARUAH. "Timing Faults and Mixed Criticality Systems". In : *Dependable and Historic Computing - Essays Dedicated to Brian Randell on the Occasion of His 75th Birthday*. 2011, p. 147–166.

[BD14] Alan BURNS et Rob DAVIS. *Mixed Criticality Systems - A Review*. Rapp. tech. Department of Computer Science, University of York, UK, 2014. URL : `http://www.cs.york.ac.uk/media/computerscience/documents/researchprojects/revie%20w2013c.pdf`.

[CRB13] Fabien CADORET, Thomas ROBERT, Etienne BORDE, Laurent PAUTET et Frank SINGHOFF. "Deterministic implementation of periodic-delayed communications and experimentation in AADL". In : *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, (ISORC'13)*. 2013, p. 1–8.

[CLB06] John M. CALANDRINO, Hennadiy LEONTYEV, Aaron BLOCK, UmaMaheswari C. DEVI et 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)*. 2006, p. 111–126.

[CGP08] Jean-Louis CAMUS, Olivier GRAFF et Sébastien POUSSARD. "A verifiable architecture for multitask, multi-rate synchronous software". In : *4th Embedded Real-Time Software Congress (ERTS'08)*. 2008.

[Car05] François CARCENAC. "Un modèle d'abstraction pour la vérification des systèmes embarqués distribués : application à l'avionique". Thèse de doctorat. SupAéro, 2005.

[CCH14] Janette CARDOSO, Jean-Charles CHAUDEMAR, Alexandre HAMEZ, Jérôme HUGUES et Pierre SIRON. "PRISE : une plate-forme de simulation distribuée pour l'ingénierie des systèmes embarqués". In : *Génie Logiciel* 108 (2014), p. 29–34.

[CPHP87] Paul CASPI, Daniel PILAUD, Nicolas HALBWACHS et John PLAICE. "Lustre : A Declarative Language for Programming Synchronous Systems". In : *POPL*. 1987, p. 178–188.

[CSST08] Paul CASPI, Norman SCAIFE, Christos SOFRONIS et Stavros TRIPAKIS. "Semantics-preserving multitask implementation of synchronous programs". In : *Trans. on Embedded Computing Sys.* 7.2 (2008), p. 1–40. ISSN : 1539-9087. DOI : `http://doi.acm.org/10.1145/1331331.1331339`.

[CRM10a] Sudipta CHATTOPADHYAY, Abhik ROYCHOUDHURY et Tulika MITRA. "Modeling shared cache and bus in multi-cores for timing analysis". In : *in 13th International Workshop on Software Compilers for Embedded Systems (SCOPES'10)*. ACM, 2010, p. 1–10. ISBN : 978-1-4503-0084-1.

[CFSV95] Thomas CHEATHAM, Amr FAHMY, Dan C. STEFANESCU et Leslie G. VALIANT. "Bulk Synchronous Parallel Computing – A Paradigm for Transportable Software". In : *In Proc. IEEE 28th Hawaii Int. Conf. on System Science*. Society Press, 1995, p. 268–275.

[CSB90] Houssine CHETTO, Maryline SILLY et T. BOUCHENTOUF. "Dynamic scheduling of real time tasks under precedence constraints". In : (1990).

[CLMW14] Denis CLARAZ, Thierry LEYDIER, Ralph MADER et Gerhard WIRRER. "Introducing Multicore at Automotive Engine Systems". In : *Embedded Real Time Software and Systems (ERTS'14)*. 2014.

[FGH06] Peter H. Feiler, David P. Gluch et John J. Hudak. *The Architecture Analysis & Design Language (AADL) : an introduction*. Rapp. tech. Carnegie Mellon University, 2006.

[CGJ96] E.G. Coffman Jr, M.R. Garey et D.S. Johnson. “Approximation algorithms for bin packing : A survey”. In : *Approximation algorithms for NP-hard problems*. PWS Publishing Co. 1996, p. 46–93.

[CP03] Jean-Louis Colaço et Marc Pouzet. “Clocks as First Class Abstract Types”. In : *Third International Conference on Embedded Software (EMSOFT'03)*. Philadelphia, USA, oct. 2003.

[CPRS03] Antoine Colin, Isabelle Puaut, Christine Rochange et Pascal Sainrat. “Calcul de majorants de pire temps d'exécution : état de l'art”. In : *Technique et Science Informatiques* 22.5 (2003), p. 651–677.

[Con08] The MISRA Consortium. *MISRA-C :2004 : Guidelines for the Use of the C Language in Critical Systems, Edition 2*. Rapp. tech. ISBN 0 9524156 2 3. MISRA, 2008.

[CHW02] Keith D. Cooper, Timothy J. Harvey et Todd Waterman. *Building a Control-Flow Graph from Scheduled Assembly Code*. Rapp. tech. TR02-399. Rice University, 2002.

[Cor12] Mikel Cordovilla Mesonero. “Environnement de développement d'applications multipériodiques sur plateforme multicoeur. La boîte à outil SchedMCore”. Thèse de doct. Toulouse, France : Université de Toulouse - ISAE/ONERA, 2012.

[CRS11] João Craveiro, José Rufino et Frank Singhoff. “Architecture, mechanisms and scheduling analysis tool for multicore time- and space-partitioned systems”. In : *SIGBED Review* 8.3 (2011), p. 23–27.

[CRM10b] Alfons Crespo, Ismael Ripoll et Miguel Masmano. “Partitioned Embedded Architecture Based on Hypervisor : The XtratuM Approach”. In : *8th European Dependable Computing Conference (EDCC'10)*. 2010, p. 67–72.

[CG06] Liliana Cucu et Joël Goossens. “Feasibility Intervals for Fixed-Priority Real-Time Scheduling on Uniform Multiprocessors”. In : *ETFA*. 2006, p. 397–404.

[CG11] Liliana Cucu-Grosjean et Joël Goossens. “Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms”. In : *Journal of Systems Architecture* 57.5 (2011), p. 561–569.

[CFG10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet et Reinhard Wilhelm. “Predictability Considerations in the Design of Multi-Core Embedded Systems”. In : *Ingénieurs de l'Automobile* 807 (sept. 2010), p. 36–42.

[Cur05] Adrian Curic. “Implementing Lustre Programs on Distributed Platforms with Real-Time Constraints”. Thèse de doct. Grenoble : Université Joseph Fourier, 2005.

[DS14] Tao Chen Daniel Lo Mohamed Ismail et G. Edward Suh. “Slack-Aware Opportunistic Monitoring for Real-Time Systems”. In : *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*. 2014.

[DILS10] Alexandre David, Jacob Illum, Kim G. Larsen et Arne Skou. “Model-Based Design for Embedded Systems”. In : CRC Press, 2010. Chap. Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1, p. 93–119. isbn : 978-1-4200-6784-2.

[DB11] Robert I. Davis et Alan Burns. “A Survey of Hard Real-time Scheduling for Multiprocessor Systems”. In : *ACM Computing Surveys (CSUR)* 43.4 (oct. 2011), 35 :1–35 :44.

[DGZS10] Marco Di Natale, Liangpeng Guo, Haibo Zeng et Alberto L. Sangiovanni-Vincentelli. “Synthesis of Multi-task Implementations of Simulink Models with Minimum Delays”. In : *IEEE Trans. Industrial Informatics* 6.4 (2010), p. 637–651.

[DWS09] Marco Di Natale, Guoqiang Wang et Alberto L. Sangiovanni-Vincentelli. “Improving the size of communication buffers in synchronous models with time constraints”. In : *IEEE Trans. Industrial Informatics* 5.3 (2009), p. 229–240.

[Dur98] Emmanuel Durand. “Description et vérification d'architectures d'application temps réel : CLARA et les réseaux de Petri temporels”. Thèse de doct. Ecole Centrale de Nantes, 1998.

[Eke04] Cecilia EKELIN. “An Optimization Framework for Scheduling of Embedded Real-Time Systems”. Thèse de doct. Chalmers University of Technology, 2004.

[Est12] ESTEREL TECHNOLOGIES, INC. *SCADE Language - Reference Manual.* 2012.

[FDT04] Sébastien FAUCOU, Anne-Marie DÉPLANCHE et Yvon TRINQUET. “An ADL Centric Approach for the Formal Design of Real-Time Systems”. In : *Architecture Description Language Workshop at IFIP World Computer Congress (WADL'04).* T. 176. Toulouse, France, 2004, p. 67–82.

[FB13] Tom FLEMING et Alan BURNS. “Extending Mixed Criticality Scheduling”. In : *Proceedings of the Workshop on Mixed Criticality (WMC'13).* 2013, p. 7–12.

[For09] Julien FORGET. “A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints”. Thèse de doct. Toulouse, France : Université de Toulouse - ISAE/ONERA, nov. 2009.

[Fre06] FREESCALE. *e600 PowerPC - Reference Manual.* 2006.

[Fre08] FREESCALE. *MPC8641D : Integrated Host Processor Family Reference Manual.* 2008.

[Fre10] FREESCALE. *QorIQ P4080 : Communications Processors with Data Path.* 2010.

[GGJY76] M. R. GAREY, Ronald L. GRAHAM, David S. JOHNSON et Andrew Chi-Chih YAO. “Resource Constrained Scheduling as Generalized Bin Packing”. In : *J. Comb. Theory, Ser. A* 21.3 (1976), p. 257–298.

[GHKT14] Pierre-Loïc GAROCHE, Falk HOWAR, Temesghen KAHSAI et Xavier THIRIOUX. “Testing-Based Compiler Validation for Synchronous Languages”. In : *NASA Formal Methods (NFM 2014).* T. 8430. Lecture Notes in Computer Science. 2014, p. 246–251.

[Gat13] Marc GATTI. “Development and certification of Avionics Platforms on Multi-Core processors”. In : *Tutorial Mixed-Criticality Systems : Design and Certification Challenges, ESWeek.* Montreal, Canada, 2013.

[GGL13] Gilles GEERAERTS, Joël GOOSSENS et Markus LINDSTRÖM. “Multiprocessor schedulability of arbitrary-deadline sporadic tasks : complexity and antichain algorithm”. In : *Real-Time Systems* 49.2 (2013), p. 171–218.

[GCS12] Clément GERVAIS, Jean-Baptiste CHAUDRON, Pierre SIRON, Régine LECONTE et David SAUSSIÉ. “Real-Time Distributed Aircraft Simulation through HLA”. In : *16th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT 2012).* 2012.

[GN03] Alain GIRAULT et Xavier NICOLLIN. “Clock-Driven Automatic Distribution of Lustre Programs”. In : *Proceedings of the 3rd International Conference on Embedded Software (EMSOFT'03).* T. 2855. Lecture Notes in Computer Science. Philadelphia, USA, 2003, p. 206–222.

[GNP06] Alain GIRAULT, Xavier NICOLLIN et Marc POUZET. “Automatic rate desynchronization of embedded reactive programs”. In : *ACM Trans. Embedded Comput. Syst.* 5.3 (2006), p. 687–717.

[Gir12a] Sylvain GIRBAL. *Flight Management System : Addendum to use-case specifications, Upper FMS part.* Rapp. tech. Thales Research & Technology, 2012.

[Gir12b] Sylvain GIRBAL. *Flight Management System Use-case overview.* Rapp. tech. Thales Research & Technology, 2012.

[Gra69] Ronald Lewis GRAHAM. “Bounds on Multiprocessing Timing Anomalies”. In : *SIAM Journal on Applied Mathematics* 17 (1969), p. 416–429.

[GLS99] Thierry GRANDPIERRE, Christophe LAVARENNE et Yves SOREL. “Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors”. In : *Proceedings of the Seventh International Workshop on Hardware/Software Codesign (CODES'99).* 1999, p. 74–78.

[GC01] Emmanuel GROLLEAU et Annie CHOQUET-GENIET. “Ordonnancement de tâches temps réel en environnement multiprocesseur à l'aide de réseaux de Petri”. In : *Real-Time Systems, RTS'2001.* Paris, France, 2001.

[GGC13] Emmanuel Grolleau, Joël Goossens et Liliana Cucu-Grosjean. “On the periodic behavior of real-time schedulers on identical multiprocessor platforms”. In : *CoRR* abs/1305.3849 (2013).

[Gro07] Object Management Group. *A UML Profile for MARTE*. Rapp. tech. Object Management Group, Inc, 2007.

[GGD07] Nan Guan, Zonghua Gu, Qingxu Deng, Shuaihong Gao et 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)*. 2007, p. 263–272.

[GGL08] Nan Guan, Zonghua Gu, Mingsong Lv, Qingxu Deng et Ge Yu. “Schedulability Analysis of Global Fixed-Priority or EDF Multiprocessor Scheduling with Symbolic Model-Checking”. In : *ISORC*. 2008, p. 556–560.

[Hal92] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. Norwell, MA, USA : Kluwer Academic Publishers, 1992. isbn : 0792393112.

[HKM12] Jonathan L. Herman, Christopher J. Kenna, Malcolm S. Mollison, James H. Anderson et Daniel M. Johnson. “RTOS Support for Multicore Mixed-Criticality Systems”. In : *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'12)*. 2012, p. 197–208.

[HCDJ08] Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Déplanche et Narendra Jussien. “Solving a real-time allocation problem with constraint programming”. In : *J. Syst. Softw.* 81.1 (jan. 2008), p. 132–149. issn : 0164-1212.

[IBM14] IBM ILOG. *CPLEX Optimization Studio*. 2014. url : `http://www.ibm.com/software/integration/optimization/cplex-optimization-studio/`.

[IEE10] IEEE. “IEEE Standard for Modeling and Simulation High Level Architecture (HLA)”. In : *IEEE Std 1516-2010* 18 (2010), p. 1–38. doi : `10.1109/IEEESTD.2010.5553440`.

[Int10] Intel Labs. *SCC External Architecture Specification (EAS)*. Rapp. tech. Intel Corporation, mai 2010.

[Int12] Intel Labs. *The SCC Programmer's Guide*. Rapp. tech. Intel Corporation, jan. 2012.

[Kah74] Gilles Kahn. “The Semantics of Simple Language for Parallel Programming”. In : *International Federation for Information Processing (IFIP'74) Congress*. New York, USA, 1974.

[Kal12] Kalray. *The MPPA hardware architecture*. 2012.

[KSB04] Christophe Kehren, Christel Seguin, Pierre Bieber, Charles Castel, Christian Bougnol, Jean-Pierre Heckmann et Sylvain Metge. “Architecture Patterns for Safe Design”. In : *AAAF 1st Complex and Safe Systems Engineering Conference*. 2004.

[Kop92] Hermann Kopetz. “Sparse time versus dense time in distributed real-time systems”. In : *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*. IEEE. 1992, p. 460–467.

[KB03] Hermann Kopetz et Günther Bauer. “The Time-Triggered Architecture”. In : *Proceedings of the IEEE* 91.1 (2003).

[Kra14] Matthew W Kracht. “Real-Time Embedded Software Modeling and Synthesis using Polychronous Data Flow Languages”. Thèse de doct. Virginia Polytechnic Institute et State University, 2014.

[LPT09] Kai Lampka, Simon Perathoner et Lothar Thiele. “Analytic real-time analysis and timed automata : a hybrid method for analyzing embedded real-time systems”. In : *Proceedings of the 9th ACM & IEEE International conference on Embedded software (EMSOFT 2009)*. 2009.

[Lap13] Jean-Claude Laperche. “Multi/many-core in Avionics Systems”. In : *4th Workshop TORRENTS*. Toulouse, France, 2013.

[LMM98] Sylvain Lauzac, Rami Melhem et 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*. 1998, p. 188–195.

[LP10] Daniel Le Berre et Anne Parrain. “The Sat4j library, release 2.2 system description”. In : *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), p. 59–64.

[LT01] Jean-Yves Le Boudec et Patrick Thiran. *Network calculus : a theory of deterministic queuing systems for the internet*. Springer-Verlag, 2001. isbn : 3-540-42184-X.

[LS96] Edward A. Lee et Alberto Sangiovanni-Vincentelli. *The tagged signal model - a preliminary version of a denotational framework for comparing models of computation*. Rapp. tech. UCB/ERL M96/33. University of California, Berkeley, CA, juin 1996.

[LS81] C.E. Leiserson et J.B. Saxe. “Optimizing synchronous systems”. In : *Foundations of Computer Science, 1981. SFCS'81. 22nd Annual Symposium on*. IEEE. 1981, p. 23–36.

[LM80] Joseph Y.-T. Leung et M. L. Merrill. “A Note on Preemptive Scheduling of Periodic, Real-Time Tasks”. In : *Inf. Process. Lett.* 11.3 (1980), p. 115–118.

[LB12] Haohan Li et Sanjoy K. Baruah. “Global Mixed-Criticality Scheduling on Multiprocessors”. In : *24th Euromicro Conference on Real-Time Systems (ECRTS'12)*. 2012, p. 166–175.

[LRSF04] Peng Li, Binoy Ravindran, Syed Suhaib et Shahrooz Feizabadi. “A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems”. In : *IEEE Trans. Softw. Eng.* 30 (9 2004), p. 613–629.

[LGG11] Markus Lindström, Gilles Geeraerts et Joël Goossens. “A faster exact multiprocessor schedulability test for sporadic tasks”. In : *19th International Conference on Real-Time and Network Systems, (RTNS'11)*. 2011, p. 25–34.

[LL73] Chung Laung Liu et James W. Layland. “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”. In : *J. ACM* 20.1 (1973), p. 46–61. issn : 0004-5411. doi : `http://doi.acm.org/10.1145/321738.321743`.

[LRL10] Isaac Liu, Jan Reineke et Edward A. Lee. “A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties”. In : *44th Asilomar Conference on Signals, Systems, and Computers*. Nov. 2010.

[MDAS10] Frédéric Mallet, Julien DeAntoni, Charles André et Robert de Simone. “The clock constraint specification language for building timed causality models”. In : *Innovations in Systems and Software Engineering* 6 (1 2010). 10.1007/s11334-009-0109-0, p. 99–106. issn : 1614-5046.

[MDB13] Renato Mancuso, Roman Dudko, Emiliano Betti, Marco Cesati, Marco Caccamo et Rodolfo Pellizzoni. “Real-time cache management framework for multi-core architectures”. In : *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*. 2013, p. 45–54.

[MH96] Florence Maraninchi et Nicolas Halbwachs. “Compositional Semantics of Non-Deterministic Synchronous Languages”. In : *Proceedings of the 6th European Symposium on Programming Languages and Systems*. ESOP '96. London, UK : Springer-Verlag, 1996, p. 235–249. isbn : 3-540-61055-3. url : `http://dl.acm.org/citation.cfm?id=645391.651455`.

[Mar04] Steven Martin. “Maîtrise de la dimension temporelle de la Qualité de Service dans les Réseaux”. Thèse de doct. Univsersité Paris XII, 2004.

[MTGL08] Hugo Metivier, Jean-Pierre Talpin, Thierry Gautier et Paul Le Guernic. “Analysis of periodic clock relations in polychronous systems”. In : *IFIP, Distributed Embedded Systems : Design, Middleware and Ressources (DIPES'08)*. Milano, Italy, sept. 2008.

[Moh98] Prasant Mohapatra. “Wormhole Routing Techniques for Directly Connected Multicomputer Systems”. In : *ACM Computing Surveys (CSUR)* 30.3 (sept. 1998), p. 374–410. issn : 0360-0300.

[MEA10] Malcolm S. Mollison, Jeremy P. Erickson, James H. Anderson, Sanjoy K. Baruah, John A. Scoredos et Northrop Grumman Corporation. “Mixed-Criticality Real-Time Scheduling for Multicore Systems”. In : *10th IEEE International Conference on Computer and Information Technology (CIT'10)*. 2010, p. 1864–1871.

[NYG13] Vincent Nelis, Patrick Meumeu Yomsi et Joël Goossens. “Feasibility Intervals for Homogeneous Multicores, Asynchronous Periodic Tasks, and FJP Schedulers”. In : *Proceedings of the 21st International Conference on Real-Time Networks and Systems*. RTNS ’13. Sophia Antipolis, France : ACM, 2013, p. 277–286. isbn : 978-1-4503-2058-0. doi : 10.1145/2516821.2516848. url : http://doi.acm.org/10.1145/2516821.2516848.

[NRS09] Eric Noulard, Jean-Yves Rousselot et Pierre Siron. “CERTI, an Open Source RTI, Why and How ?” In : *Spring Simulation Interoperability Workshop* (mar. 2009), p. 23–27.

[NP12] Jan Nowotsch et Michael Paulitsch. “Leveraging Multi-core Computing Architectures in Avionics”. In : *Ninth European Dependable Computing Conference (EDCC’12)*. 2012, p. 132–143.

[NP13] Jan Nowotsch et Michael Paulitsch. “Quality of service capabilities for hard real-time applications on multi-core processors”. In : *21st International Conference on Real-Time Networks and Systems (RTNS’13)*. 2013, p. 151–160.

[NPB14] Jan Nowotsch, Michael Paulitsch, Daniel Bühler, Henrik Theiling, Simon Wegener et Michael Schmidt. “Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement”. In : *26th Euromicro Conference on Real-Time Systems (ECRTS’14)*. 2014.

[OMG10] OMG. *Systems Modeling Language*. Rapp. tech. Object Management Group, Inc, 2010.

[OTS99] Timothy W. O’Neil, Sissades Tongsima et Edwin H.M. Sha. “Optimal scheduling of data-flow graphs using extended retiming”. In : *Proceedings of the ISCA 12th International Conference on Parallel and Distributed Computing Systems*. 1999, p. 292–297.

[OSE05] OSEK. *OSEK/VDX - Operating System - Version 2.2.3*. Rapp. tech. OSEK Group, 2005.

[OTE12] Martin Otter, Bernhard Thiele et Hilding Elmqvist. “A library for synchronous control systems in modelica”. In : *Proceedings of 9th International Modelica Conference, Munich, Germany, September*. 2012, p. 3–5.

[Pat12] Risat Mahmud Pathan. “Schedulability Analysis of Mixed-Criticality Systems on Multiprocessors”. In : *24th Euromicro Conference on Real-Time Systems (ECRTS’12)*. 2012, p. 309–320.

[PBB11] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo et Russell Kegley. “A Predictable Execution Model for COTS-based Embedded Systems”. In : *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2011)*. 2011.

[PSA97] Dar-Tzen Peng, Kang G. Shin et Tarek F. Abdelzaher. “Assignment and Scheduling Communicating Periodic Tasks in Distributed Real-Time Systems”. In : *IEEE Trans. Software Eng.* 23.12 (1997), p. 745–758.

[PQB14] Luis Miguel Pinho, Eduardo Quiñones, Marko Bertogna, Andrea Marongiu, Jorge Pereira Carlos, Claudio Scordino et Michele Ramponi. “P-SOCRATES : a Parallel Software Framework for Time-Critical Many-Core Systems”. In : *Euromicro Conference on Digital System Design (DSD’14)*. 2014.

[Pou13] Luis-Noel Pouchet et al. *PolyBench Benchmark Suite*. http://www.cse.ohio-state.edu/~pouchet/software/polybench/. 2013.

[Pto14] Claudius Ptolemaeus, éd. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. url : http://ptolemy.org/books/Systems.

[RGR09] Ahmed Rahni, Emmanuel Grolleau et Michaël Richard. “An efficient response-time analysis for real-time transactions with fixed priority assignment”. In : *ISSE* 5.3 (2009), p. 197–209.

[RH02] Mario A. Rivas et Michael G. Harbour. “POSIX-Compatible Application-Defined Scheduling in MaRTE OS”. In : *14th Euromicro Conference on Real-Time Systems*. 2002.

[RDK00] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter R. Mattson et John D. Owens. “Memory access scheduling”. In : *in 27th International Symposium on Computer Architecture (ISCA’00)*. 2000, p. 128–138.

[Roc11] Christine Rochange. “Prévisibilité des temps d’exécution pire-cas”. français. Habilitation à diriger des recherches. Toulouse, France : Université de Toulouse, nov. 2011.

[RvW06] Francesca Ross, Peter van Beek et Toby Walsh, éds. *Handbook of Constraint Programming*. Elsevier, 2006.

[RTC08] RTCA, Inc. *DO-178 ED-12B - Software Considerations in Airborne Systems and Equipment Certification*. 2008.

[RTC11a] RTCA, Inc. *DO-178 ED-12C - Software Considerations in Airborne Systems and Equipment Certification*. 2011.

[RTC11b] RTCA, Inc. *DO-333 - Formal Methods Supplement to DO-178C and DO-278A*. 2011.

[RDS14] Stéphane Rubini, Pierre Dissaux et Frank Singhoff. “Modeling Shared-Memory Multiprocessor Systems with AADL”. In : *Proceedings of the First International Workshop on Architecture Centric Virtual Integration co-located with the 17th International Conference on Model Driven Engineering Languages and Systems (ACVI@MoDELS’14)*. 2014.

[RFKK08] Ana-Elena Rugina, Peter H. Feiler, Karama Kanoun et Mohamed Kaâniche. “Software dependability modeling using an industry-standard architecture description language”. In : *Embedded Systems and Real-Time Systems (ERTS’08)*. 2008.

[SAE10] SAE. *Aerospace Recommended Practices ARP4754a - Development of Civil Aircraft and Systems*. SAE. 2010.

[Sch12] Johannes Scheller. “Real-time operating systems for many-core platforms”. Mém.de mast. Toulouse, France : ISAE/ONERA, 2012.

[SW00] Klaus Schild et Jörg Würtz. “Scheduling of Time-Triggered Real-Time Systems”. In : *Constraints* 5.4 (oct. 2000), p. 335–357. issn : 1383-7133.

[SCT10] Andreas Schranzhofer, Jian-Jia Chen et Lothar Thiele. “Timing Analysis for TDMA Arbitration in Resource Sharing Systems”. In : *16th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2010)*. Stockholm, Sweden, 2010.

[ST98] David B. Skillicorn et Domenico Talia. “Models and Languages for Parallel Computation”. In : *ACM Computing Surveys* 30 (1998), p. 123–169.

[SL97] Irina Smarandache et Paul Le Guernic. *A Canonical Form for Affine Relations in Signal*. Rapp. tech. RR-3097. INRIA, 1997.

[STC06] Christos Sofronis, Stavros Tripakis et Paul Caspi. “A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling”. In : *EMSOFT ’06 : Proceedings of the 6th ACM & IEEE International conference on Embedded software*. Seoul, Korea : ACM, 2006, p. 21–33. isbn : 1-59593-542-8. doi : `http://doi.acm.org/10.1145/1176887.1176892`.

[Sor96] Yves Sorel. “Real-time embedded image processing applications using the A^3 methodology”. In : *ICIP (2)*. 1996, p. 145–148.

[Spa12] Jens Sparso. “Design of Networks-on-Chip for Real-Time Multi-processor Systems-on-Chip”. In : *Proceedings of the 2012 12th International Conference on Application of Concurrency to System Design*. ACSD ’12. 2012, p. 1–5. isbn : 978-0-7695-4709-1.

[TBB13] Julien Tanguy, Jean-Luc Béchennec, Mikaël Briday, Sebastien Dube et Olivier H. Roux. “Device driver synthesis for embedded systems”. In : *Proceedings of 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation, (ETFA’13)*. 2013, p. 1–8.

[Tar73] Robert Endre Tarjan. “Enumeration of the Elementary Circuits of a Directed Graph”. In : *SIAM J. Comput.* 2.3 (1973), p. 211–216.

[Tex13] Texas Instruments. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor*. Rapp. tech. SPRS691D. Texas Instruments Incorporated, 2013.

[THA10] THALES. *FMS 220 Software Requirement Specification (SRS)*. Rapp. tech. THALES, 2010, p. 1–1019.

[The14] The Mathworks. *Simulink : User’s Guide*. The Mathworks. 2014.

[TCN00] Lothar Thiele, Samarjit Chakraborty et Martin Naedele. "Real-Time Calculus For Scheduling Hard Real-Time Systems". In : *IEEE International Symposium on Circuits and Systems (ISCAS)*. 2000, p. 101–104.

[Til13a] Tilera Corp. *Tile processor architecture - Overview for the TILEPro Series*. Rapp. tech. UG120. 2013.

[Til13b] Tilera Corp. *Tilera Documentation : Gx MDE Programming Overview*. Rapp. tech. UG 505. 2013.

[TSCC05] Stavros Tripakis, Christos Sofronis, Paul Caspi et Adrian Curic. "Translating Discrete-time Simulink to Lustre". In : *ACM Trans. Embed. Comput. Syst.* 4.4 (2005), p. 779–818.

[UCS10] 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, Jörg Mische, Marco Paolieri et Julian Wolf. "MERASA : Multi-Core Execution of Hard Real-Time Applications Supporting Analysability". In : *IEEE Micro* 30.5 (sept. 2010), p. 66–75.

[Ves07] Steve Vestal. "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance". In : *Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07)*. 2007, p. 239–243.

[WEE08] 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 et Per Stenström. "The worst-case execution-time problem—overview of methods and survey of tools". In : *ACM Trans. Embed. Comput. Syst.* 7.3 (mai 2008), 36 :1–36 :53. issn : 1539-9087.

[WR12] Reinhard Wilhelm et Jan Reineke. "Embedded Systems : Many Cores – Many Problems". In : *Symposium on Industrial Embedded Systems (SIES'12)*. 2012.

[Xu93] Jia Xu. "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations". In : *IEEE Transactions on Software Engineering* 19.2 (1993), p. 139–154.

[XP93] Jia Xu et David Lorge Parnas. "On Satisfying Timing Constraints in Hard-Real-Time Systems". In : *IEEE Transaction on Software Engineering* 19.1 (1993), p. 70–84.

[YKRB14] Eugene Yip, Matthew Kuo, Partha Roop et David Broman. "Relaxing the Synchronous Approach for Mixed-Criticality Systems". In : *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*. Avr. 2014.

[YRBG13] Eugene Yip, Partha Roop, Morteza Biglari-Abhari et Alain Girault. "Programming and Timing Analysis of Parallel Programs on Multicores". In : *International Conference on Application of Concurrency to System Design, ACSD'13*. IEEE, 2013, p. 167–176.

[YMWP14] Heechul Yun, Renato Mancuso, Zheng-Pei Wu et Rodolfo Pellizzoni. "PALLOC : DRAM Bank-Aware Memory Allocator for Performance Isolation on Multicore Platforms". In : *20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*. Avr. 2014.

[YYP13] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo et Lui Sha. "MemGuard : Memory bandwidth reservation system for efficient performance isolation in multi-core platforms". In : *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'13)*. 2013, p. 55–64.

[ZD12] Haibo Zeng et Marco Di Natale. "Schedulability Analysis of Periodic Tasks Implementing Synchronous Finite State Machines". In : *24th Euromicro Conference on Real-Time Systems (ECRTS 2012)*. IEEE Computer Society, 2012, p. 353–362.

[ZB12] Michael Ziwisky et Dennis Brylow. "BareMichael : A Minimalistic Bare-metal Framework for the Intel SCC". In : *Proceedings of the 6th Many-core Applications Research Community (MARC) Symposium*. 2012, p. 66–71.